safecreative
0903012663983
INFO ABOUT RIGHTS

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA UNIVERSITARIA DE INFORMÁTICA

TRABAJO FIN DE CARRERA

EMULADOR DE SEGA MASTER SYSTEM EN C++

DIRECTOR: JOSÉ GABRIEL PÉREZ DÍEZ

AUTOR: ISRAEL LÓPEZ FERNÁNDEZ

CONTACTO: ilopez@retrowip.com

http://www.retrowip.com

FEBRERO 2009

Licencia de uso

El presente documento se encuentra licenciado por su autor, Israel López Fernández, bajo una licencia de tipo *Creative Commons* "Reconocimiento-No comercial-Compartir bajo la misma licencia 3.0 España".

Bajo los términos de esta licencia:

Usted es libre de:

Copiar, distribuir y comunicar públicamente la obra.

Hacer obras derivadas.

Bajo las condiciones siguientes:

Reconocimiento. Debe reconocer y atribuir esta obra, "Emulador de SEGA Master System en C++", a Israel López Fernández (pero no de una manera que sugiera que tiene su apoyo o apoya el uso que hace de su obra), así como incluir un enlace a la página web del autor (http://www.retrowip.com).

No comercial. No puede utilizar esta obra para fines comerciales.

Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Nada en esta licencia menoscaba o restringe los derechos morales del autor.

Índice de contenidos

Índice de contenidos..... - 1 -

Listado de figuras..... - 5 -

Capítulo 1. Introducción. ..... - 7 -

1. Motivación y propósito del estudio. ..... - 7 -
2. ¿Qué se entiende por emulador? ..... - 8 -
3. Pequeña historia de la emulación..... - 9 -
4. La máquina a emular: SEGA Master System. ..... - 9 -
5. Objetivos iniciales. ..... - 11 -
6. Estructuración del documento. ..... - 11 -

Capítulo 2. Visión global de la SEGA Master System. ..... - 13 -

1. Visión hardware. ..... - 13 -
2. Implementación. ..... - 14 -

Capítulo 3. La unidad central de proceso (CPU). ..... - 15 -

1. Arquitectura del Z80. ..... - 15 -
2. Interrupciones en el Z80. ..... - 17 -
3. Formato de instrucciones del Z80..... - 19 -
4. Grupos de instrucciones. ..... - 20 -
5. El ciclo de instrucción..... - 21 -
6. Inicio y reinicio de la CPU. ..... - 22 -
7. Implementación. ..... - 22 -

Capítulo 4. El sistema de entrada/salida y memoria. ..... - 29 -

1. Espacios direccionables independientes. ..... - 29 -
2. Mapa de memoria..... - 29 -
3. Los mapeadores de memoria. ..... - 31 -
4. Los registros de marco. ..... - 33 -
5. Los puertos de entrada/salida. ..... - 33 -

6. Implementación. ..... - 34 -

Capítulo 5. El procesador de video (VDP). ..... - 37 -

1. Descripción general ..... - 37 -
2. Caracteres, *tiles* y *sprites.* ..... - 37 -
3. Las paletas. ..... - 38 -
4. El Generador de Caracteres. ..... - 39 -
5. El Mapa de Pantalla. ..... - 41 -
6. La Tabla de Atributos de *Sprites*. ..... - 42 -
7. Prioridades entre capas. ..... - 45 -
8. Acceso al VDP. ..... - 46 -
9. Registros del VDP. ..... - 47 -
10. Interrupciones en el VDP. ..... - 50 -
11. Implementación. ..... - 51 -

Capítulo 6. El generador de sonido programable (PSG). ..... - 59 -

1. Descripción general ..... - 59 -
2. Los canales del PSG. ..... - 60 -
3. Registros y programación del PSG. ..... - 61 -
4. Implementación. ..... - 63 -

Capítulo 7. Los dispositivos controladores. ..... - 67 -

1. Descripción general ..... - 67 -
2. Puertos de entrada/salida. ..... - 69 -
3. Implementación. ..... - 69 -

Capítulo 8. El selector de programas. ..... - 71 -

1. Introducción y parámetros. ..... - 71 -
2. Teclas de función. ..... - 72 -
3. Implementación. ..... - 74 -

Capítulo 9. Desarrollo, verificación y pruebas ..... - 75 -

Capítulo 10. Conclusiones. ..... - 79 -

1. Objetivos cumplidos..... - 79 -
2. Perspectiva..... - 79 -

Bibliografía..... - 81 -

1. Libros consultados..... - 81 -
2. Vínculos de Internet..... - 81 -

Agradecimientos..... - 83 -

Apéndice A. La biblioteca QT4..... - 85 -

Apéndice B. Las bibliotecas SDL..... - 87 -

Apéndice C. El simulador Z80Sim..... - 89 -

Apéndice D. Instrucciones detalladas del Z80..... - 93 -

1. Acrónimos y notación..... - 93 -
2. Grupo de instrucciones de carga de 8 bits..... - 94 -
3. Grupo de instrucciones de carga de 16 bits..... - 95 -
4. Grupo de instrucciones Aritméticas y Lógicas de 8 bits..... - 96 -
5. Grupo de instrucciones Aritméticas de 16 bits..... - 97 -
6. Grupo de instrucciones Aritméticas de Propósito General y Control de CPU. - 97 -
7. Grupo de instrucciones de Búsqueda, Intercambio y Transferencia..... - 98 -
8. Grupo de instrucciones de Rotación y Desplazamiento..... - 99 -
9. Grupo de instrucciones de Manipulación de bits..... - 100 -
10. Grupo de instrucciones de Entrada / Salida..... - 101 -
11. Grupo de instrucciones de Salto..... - 102 -
12. Grupo de instrucciones de Llamada y Retorno..... - 102 -

Código fuente..... - 103 -

1. Introducción..... - 103 -

Listado de figuras

Fig. 1. SEGA Master System ..... - 10 -
Fig. 2. Sega Master System II ..... - 10 -
Fig. 3. Diagrama de clases del sistema a desarrollar ..... - 14 -
Fig. 4. Arquitectura del Z80 ..... - 15 -
Fig. 5. Clase Z80 ..... - 23 -
Fig. 6. Estructura contextoZ80 ..... - 23 -
Fig. 7. Rutina de tratamiento de interrupción ..... - 24 -
Fig. 8. Ciclo de instrucción ..... - 25 -
Fig. 9. Función *ejecutaInst* ..... - 26 -
Fig. 10. Macros de instrucciones de ejemplo ..... - 26 -
Fig. 11. Reset ..... - 27 -
Fig. 12. Mapa de memoria ..... - 30 -
Fig. 13. Mapeadores de memoria ..... - 32 -
Fig. 14. Registros de marco ..... - 33 -
Fig. 15. Puertos de entrada/salida ..... - 34 -
Fig. 16. Clase PIOSMS ..... - 35 -
Fig. 17. Clase MemSMS ..... - 35 -
Fig. 18. Representación gráfica de un carácter ..... - 39 -
Fig. 19. Carácter de ejemplo ..... - 40 -
Fig. 20. Mapa de Pantalla ..... - 41 -
Fig. 21. Organización de Tabla de Atributos de *Sprites* ..... - 43 -
Fig. 22. Tabla de Atributos de *Sprites* ..... - 44 -
Fig. 23. Prioridades entre capas ..... - 45 -
Fig. 24. Comandos del VDP ..... - 46 -
Fig. 25. Clase VDP ..... - 52 -
Fig. 26. Estructura contextoVDP ..... - 53 -
Fig. 27. Tipos de reflejados ..... - 54 -
Fig. 28. Tipos de escalados ..... - 55 -
Fig. 29. Video sin filtrar ..... - 57 -
Fig. 30. Video con filtrado *raw* ..... - 57 -
Fig. 31. Video con filtrado *scanlines* ..... - 58 -
Fig. 32. Video con filtrado bilinear ..... - 58 -
Fig. 33. Diagrama de bloques del PSG ..... - 59 -
Fig. 34. Cálculo de la frecuencia del PSG ..... - 60 -
Fig. 35. Byte de *latch*/datos ..... - 62 -
Fig. 36. Byte de datos ..... - 62 -
Fig. 37. Valor de los atenuadores ..... - 63 -

Fig. 38. Clase PSG ..... - 63 -
Fig. 39. Estructura contextoPSG ..... - 64 -
Fig. 40. Función *generaMuestras* ..... - 66 -
Fig. 41. Mando de control estándar de la Master System ..... - 67 -
Fig. 42. Conexiones de los mandos de control ..... - 68 -
Fig. 43. Otros dispositivos de control ..... - 68 -
Fig. 44. Puertos de entrada/salida de los dispositivos controladores ..... - 69 -
Fig. 45. Clase Joysticks ..... - 70 -
Fig. 46. Estado interno de la clase Joysticks ..... - 70 -
Fig. 47. Interfaz gráfica ..... - 71 -
Fig. 48. Teclas de función y atajos de teclado ..... - 73 -
Fig. 49. Algunas líneas de salida de ZEXALL mostrando errores detectados ..... - 75 -
Fig. 50. Varias demos gráficas para verificar el VDP ..... - 76 -
Fig. 51. Demos para verificar controladores y mapeadores de memoria ..... - 77 -
Fig. 52. Demo “SMS Sound Test” para verificar el PSG ..... - 77 -
Fig. 53. Simulador Z80Sim ..... - 89 -
Fig. 54. Estructura de archivos del código fuente. ..... - 104 -

Capítulo 1. Introducción.

1. Motivación y propósito del estudio.

Sería allá por el año 1990, contando con 9 años, cuando tuve mi primera videoconsola, la SEGA Master System.

Mis anteriores experiencias con el mundo de los videojuegos consistían en cortas partidas esporádicas en las máquinas *arcade* de los bares o salones recreativos, con la primeriza videoconsola Philips Videopac que teníamos en casa o una clónica de la Atari 2600 que poseía mi vecino.

Con estas experiencias previas, la llegada de la Master System y su superioridad técnica supuso una fuente de diversión que hasta el día de hoy no he vuelto a repetir con ningún otro sistema de videojuegos.

Con el paso del tiempo y la popularización de los ordenadores personales fueron apareciendo los conocidos como emuladores, programas que reproducían el comportamiento de otras máquinas, y con ellos la posibilidad de volver a jugar a "mi" Master System de nuevo.

Estos emuladores durante años los consideré, desde mi desconocimiento, obras complejísimas y a sus autores auténticos gurús de la informática. Pero a medida que iba leyendo información sobre ellos, deteniéndome a pensar en su funcionamiento o estudiando materias relativas a los mismos, fui dándome cuenta de que en el fondo no era algo tan complejo e incluso yo, si me lo proponía, podría llegar a crear uno.

Quizá este interés en los propios emuladores fue el que me llevó a prestarle un interés especial en mis años universitarios a todo lo relativo a la arquitectura de computadores e intérpretes.

Una vez acabadas las asignaturas de la carrera, para la realización de este trabajo final de carrera mi primera opción era obvia: quería intentar hacer mi propio emulador. Había montones de máquinas para emular, pero mi elección era igualmente evidente, ¿qué mejor máquina que aquella con la que pasé tan buenos momentos y a la que tanto cariño le guardo?

2. ¿Qué se entiende por emulador?

Un emulador en términos de informática es un programa que permite ejecutar en una máquina, de una cierta arquitectura hardware, software de otra máquina con una arquitectura diferente.

En este momento habría que hacer una diferenciación con el término simulador, a menudo utilizados incorrectamente de forma análoga.

Tanto un simulador como un emulador pretenden el mismo objetivo: producir una salida lo más fielmente posible al original, pero la diferencia radica en cómo conseguir dicho fin. Mientras que un simulador solo se preocupa de que la salida sea la misma (a menudo olvidando cómo se llega a ella en la máquina original), un emulador prestará un especial interés en reproducir el comportamiento de la máquina original, recreando este comportamiento de la forma más fiel posible para que, a través de él, surja por sí misma la salida esperada.

Los emuladores están íntimamente relacionados con los intérpretes y las máquinas virtuales. Allí donde un intérprete interpreta un lenguaje de programación "inventado", un emulador interpretará los programas binarios escritos para una máquina física real. Por otro lado, el concepto de máquina virtual es el mismo que el de emulador, pues ambos virtualizan el comportamiento de una máquina, solo diferenciándose en que la máquina a emular en el segundo caso existe físicamente y en el primero es una representación teórica.

La principal utilidad de los emuladores es la de ahorrar costes en la migración de sistemas. Al cambiar de plataforma hardware los programas escritos para la antigua, con mucha probabilidad, serán incompatibles con la nueva. Para poder seguir usando dichos programas haría falta una recompilación de los mismos, pero muchas veces esto no es posible (por haberse perdido el código fuente original, haber desaparecido la empresa que lo creara, etc.), es muy costoso o directamente impracticable el preparar miles de programas. Aquí entra en juego el emulador, pues gracias a él se podría ejecutar cualquiera de estos programas como si se siguiera usando la máquina original. Ejemplos de esta utilización de los emuladores serían Rosetta con el paso de Apple de arquitectura PowerPC a Intel x86 o el emulador incorporado en las PDAs Palm cuando pasaron de usar arquitectura Motorola 68000 a ARM.

La otra gran utilidad, menospreciada por mucha gente por su punto de vista más nostálgico, es la de la preservación. Con el tiempo las máquinas se van quedando cada vez más obsoletas y terminan acabando en museos, en el mejor de los casos, o vertederos, en el más habitual, pero hay programas que han marcado historia en dichas máquinas y "morirán" con ellas. Aquí entran en juego los emuladores de ordenadores o videoconsolas, con los cuales los programas y juegos de dichas máquinas podrán

seguir usándose pese a su desaparición. Quizá el ejemplo más importante de este tipo de emuladores sea el proyecto MAME (*Multi Arcade Machine Emulator*) el cual persigue emular todas las máquinas *arcade* para su conservación.

3. Pequeña historia de la emulación.

Como se ha comentado anteriormente, la necesidad del primer emulador apareció cuando el primer ordenador fue reemplazado por otro incompatible y se quiso mantener la compatibilidad con el software existente. Esto se produjo en los años 60 con los primeros IBM.

En los siguientes años, a finales de los 80 y principios de los 90, con el auge de las máquinas RISC se vio la necesidad de adaptar el software existente para las anteriores máquinas CISC. Aquí imperaron los traductores binarios (la mayoría de ellos funcionando en tiempo estático, no en tiempo real), pero se siguió estudiando el tema de los emuladores.

Los primeros emuladores como se conocen actualmente aparecieron en los años 90, cuando se desarrollaron emuladores de sistemas como el PC o Commodore 64 para los potentes Amiga. Sin embargo, la diferencia de potencia de cálculo entre estas máquinas no era lo bastante grande para conseguir una emulación completa en tiempo real (por lo general se estima que para emular una máquina en tiempo real hace falta otra máquina 10 veces más potente que aquella).

Más tarde, a mediados de los 90, apareció la conocida como *emuscene*. En este movimiento muchos programadores aficionados empezaron a crear emuladores y ofrecerlos libremente para cualquier persona interesada en su uso. Este movimiento ha seguido popularizándose hasta el día de hoy, aprovechando la vertiginosa velocidad a la que ha aumentado la potencia del hardware para pasar de emular viejas máquinas de 8 bits a otras mucho más complejas de hasta 64 bits.

Finalmente, los emuladores han evolucionado a ramas comerciales como los procesadores Transmeta que a través de una capa de emulación hardware permiten ejecutar software escrito para arquitecturas Intel x86 en su arquitectura VLIW.

4. La máquina a emular: SEGA Master System.

La videoconsola Master System fue desarrollada en 1986 por la compañía japonesa SEGA para intentar dominar el mundo de los videojuegos.

Todo el sistema estaba basado en el célebre procesador Zilog Z80 (uno de los más usados de la historia y presente incluso a día de hoy en una versión miniaturizada) al que se le añadió un procesador gráfico y otro de sonido fabricados por Texas Instruments para liberar a la CPU de esta carga.

Fig. 1. SEGA Master System

Los programas escritos para esta máquina, en su mayoría juegos o software educativo, se vendían escritos en las memorias ROM incluidas en los cartuchos diseñados para la misma, incorporando la Master System tan solo una pequeña BIOS para inicializar el hardware y dar paso a estos programas.

Si bien su popularidad en su país de origen y Estados Unidos no fue demasiado grande, debido a la dura competencia existente con la videoconsola NES de Nintendo (la tradicional compañía rival de SEGA), sí obtuvo una posición preferente en Europa y, sobretodo, Brasil donde aun a día de hoy, más de 20 años después de su lanzamiento, se sigue vendiendo licenciada a la compañía Tec Toy con gran éxito entre el público.

Desde su lanzamiento se produjeron pequeñas variaciones en el hardware para abaratar costes a costa de eliminar funciones prácticamente no utilizadas. Como consecuencia de estos cambios apareció la Master System II y la versión portátil (tomando prácticamente el mismo hardware y añadiendo una pantalla y alimentación por baterías) llamada Game Gear.

Fig. 2. Sega Master System II

5. Objetivos iniciales.

El propósito de este trabajo final de carrera es implementar un emulador de Master System desde cero.

Se usará una aproximación didáctica al mismo, más centrada en la reusabilidad y encapsulación del código que en la eficiencia del mismo, pero sin renunciar a las características añadidas que suelen llevar estos programas sobre las máquinas a emular.

Entre estas característica se pretende implementar filtros de suavizado de la imagen, reescalados de la misma, posibilidad de guardar y cargar el estado actual de la máquina para continuar con el juego en otro momento (los conocidos como *savestates*), posibilidad de jugar en red, configuración de la calidad del sonido, etc.

Por otro lado se creará una interfaz gráfica para la carga, configuración y ejecución de programas de forma sencilla mediante el uso del ratón.

Finalmente, se desarrollarán las herramientas necesarias para la verificación de cada componente desarrollado, centrándose especialmente en un intérprete gráfico del procesador Z80 que haga uso de la clase desarrollada para implementar a éste.

6. Estructuración del documento.

Con el presente documento se pretende recopilar, en la medida de lo posible, toda la documentación técnica sobre la Master System y su hardware que pueda resultar de utilidad para la creación de un emulador de dicha máquina, así como sugerir unas pautas para su implementación en un lenguaje orientado a objetos como es C++.

El documento se dividirá en una serie de capítulos, uno por cada componente significativo de la Master System, en cada uno de los cuales se expondrá la información relativa a dicho componente y de qué forma utilizar estos datos para la realización de un emulador. Finalmente, se dedicará un último apartado de cada capítulo para mostrar detalles específicos de la implementación realizada, el diagrama de la clase software resultante y mostrando fragmentos de código para aquellos casos con una especial dificultad de realización o comprensión.

Al final del documento se incluirá una serie de apéndices con información específica sobre bibliotecas y software utilizado en el desarrollo del emulador que si bien no interviene en el proceso de emulación sí se ha considerado una parte relevante por apoyarse en éstos para mostrar los resultados, facilitar su uso o bien depurarlo.

Finalmente, como último apartado, se incluirá todo el código fuente desarrollado para este trabajo final de carrera.

Capítulo 2. Visión global de la SEGA Master System.

1. Visión hardware.

La Master System alberga dentro de su carcasa una serie de procesadores de propósito específico junto a un conjunto de elementos hardware discretos para acondicionar las señales de comunicación entre ellos.

El “cerebro” de la máquina es un microprocesador Z80A, fabricado por Zilog, conectado a un reloj externo con una frecuencia de 3,579545 MHz Este procesador posee un bus de datos de 8 bits y otro de direcciones de 16 bits, por lo que es capaz de direccionar hasta 64 Kbytes de memoria.

Para liberar de carga a este procesador se dispone de otros dos procesadores de propósito específico: el procesador de video y el generador de sonido.

El procesador de video es una variante del TMS9918 de Texas Instruments. Este procesador, como se verá en su capítulo dedicado, es capaz de realizar varios efectos por hardware (sin necesidad de cargar al Z80 con este trabajo) tales como escalados, dos capas de dibujado, detección de colisiones de *sprites*, etc. y proporciona una resolución de salida estándar de 256x192 píxeles. Se comunica con el Z80 por medio de unos puertos pertenecientes a su espacio de entrada/salida.

Por su parte, el generador de sonido se corresponde con el chip SN76489 también fabricado por Texas Instruments. Este chip es capaz de generar tres canales independientes de sonido y un cuarto canal especializado en producir ruidos. La salida de cada uno de estos canales es modificable por medio del periodo de la onda cuadrada asociada a cada canal, así como un atenuador por canal que le permite variar el volumen del mismo.

Si bien la videoconsola solo incorpora 8 Kbytes de memoria RAM, en los cartuchos de los juegos se incorpora una memoria ROM de hasta un Mbyte que será direccionada por medio de los *mapeadores* de memoria.

Finalmente, de forma externa a la propia videoconsola es posible conectar una serie de dispositivos controladores para interactuar con la misma. Entre estos dispositivos se encuentran mandos de control, pistolas de luz, gafas de visión tridimensional y otros dispositivos variados.

2. Implementación.

Para la implementación de la SEGA Master System se ha decidido utilizar una solución orientada a objetos empleando el lenguaje de programación C++.

Cada uno de los componentes de la máquina original (procesador principal, procesador de video, generador de sonido, sistema de memoria, sistema de entrada/salida y dispositivo controlador) será implementado mediante una clase (Z80, VDP, PSG, MemSMS, PIOSMS y Joystick, respectivamente) que dará lugar a su correspondiente objeto.

Por medio de esta modularización y encapsulado de cada componente en una clase, se pretende dotar a cada una de ellas de una cierta independencia respecto al resto de componentes. De esta manera cada componente software podría ser usado en otros proyectos diferentes para implementar emuladores de otras máquinas con las que se comparta hardware, maximizando así la reutilización de código y minimizando los esfuerzos de una nueva implementación.

Por otro lado, para las partes específicas para su ejecución en un PC se ha elegido el uso de la librería multimedia multiplataforma SDL, de forma que el software resultante sea fácilmente portable a cualquier otra plataforma o sistema operativo, pero separándose su uso de las propias clases de cada componente, de forma que pudiera sustituirse por otro tipo de librerías multimedia sin afectar a las clases ya codificadas.

En la siguiente figura se puede ver un diagrama de clases del modelo completo a ser implementado.

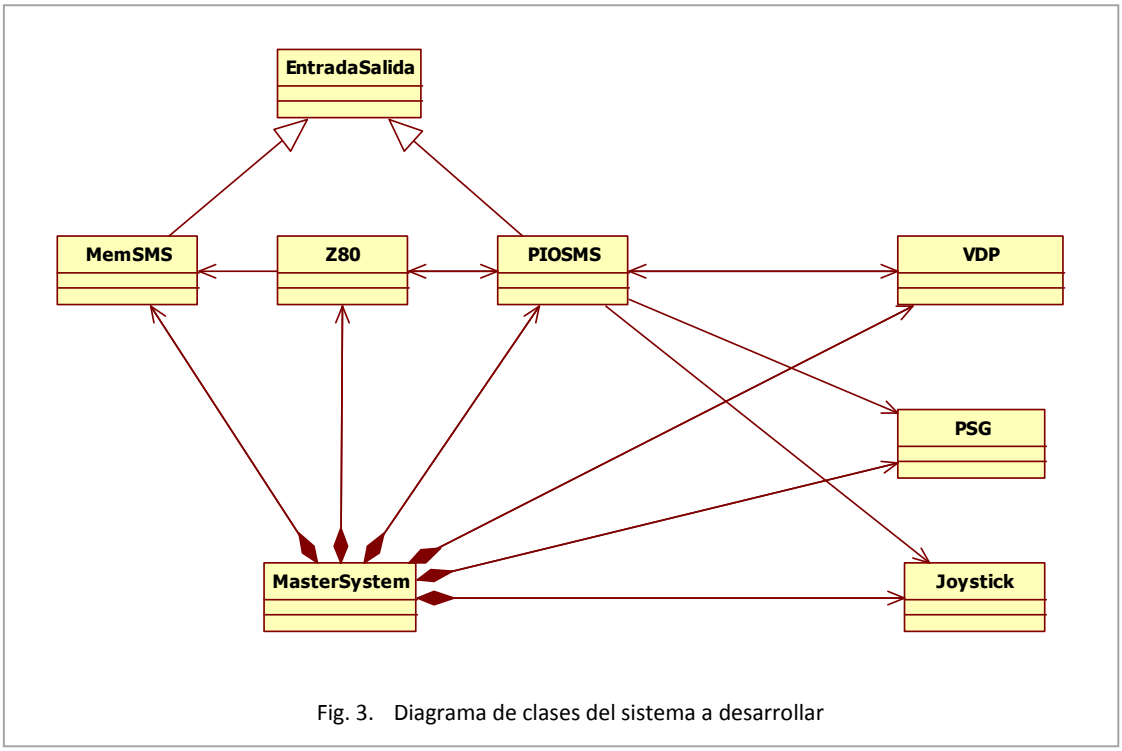


Fig. 3. Diagrama de clases del sistema a desarrollar

Capítulo 3. La unidad central de proceso (CPU).

1. Arquitectura del Z80.

El auténtico “cerebro” de la Master System es su unidad central de proceso, concretamente un microprocesador Z80A fabricado por Zilog.

Éste es un procesador CISC diseñado para ser compatible con los programas escritos para el procesador 8080 de Intel, de forma que comparte el juego de instrucciones de aquél (con los mismos códigos de operación) y además añade varias instrucciones más para nuevos programas que no tengan que mantener la compatibilidad.

En el caso de la Master System está conectado a un reloj externo de 3,579545 MHz que será el que lleve la temporización del resto de componentes del hardware.

A continuación se muestra un esquema de su arquitectura:

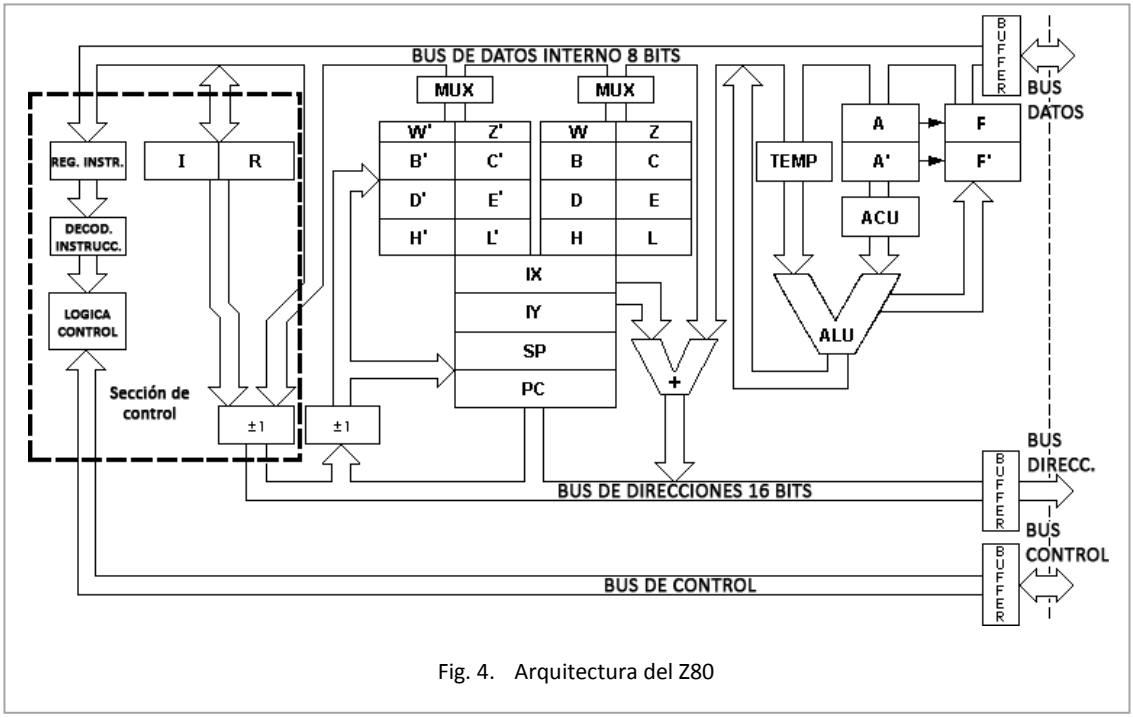


Fig. 4. Arquitectura del Z80

Como se puede observar, este procesador incorpora un bus de datos de 8 bits, limitando el tamaño de los datos intercambiados con los periféricos y memoria a un byte, y un bus de direcciones de 16 bits que supondrá un máximo de 65.536 palabras de memoria direccionables.

Por otro lado, tiene dos espacios de direccionamiento: de memoria y de entrada/salida, pudiendo direccionar la misma cantidad de unidades en cada uno de ellos.

Del mismo gráfico se puede observar que existen 26 registros diferentes:

- B, C, D, E, H, L y B', C', D', E', H', L'. Estos registros son de 8 bits, pero pueden agruparse por pares en registros de 16 bits de la forma BC, DE, HL y B'C', D'E', H'L'. Tienen la particularidad de que solo son accesibles a la vez 6 de ellos, pues están divididos en 2 bancos intercambiables por medio de unas instrucciones específicas de cambio de banco.
- A, F y A', F'. Como en el caso anterior se trata de registros de 8 bits de los que solo dos son accesibles al mismo tiempo por estar divididos en dos bancos. El registro A es el acumulador que siempre será uno de los operandos de la Unidad Aritmético Lógica (ALU). Por su parte, en el registro F se encuentran los flags de la máquina que guardan su información de estado.
- PC. Contador de programa, apunta a la dirección de la siguiente instrucción a ejecutar, por lo tanto tiene que ser de 16 bits.
- SP. Puntero de pila, apunta a la primera dirección libre de la pila de ejecución, por tanto es necesario que también sea de 16 bits.
- IX e IY. Estos dos registros de 16 bits se usan como dirección base para instrucciones que hacen uso de vectores. Existe una mini ALU que solo sirve para sumarle un dato de 8 bits a estos registros y volcar el resultado al bus de direcciones.
- I. Este registro de 8 bits se usa para el tratamiento de interrupciones.
- R. Este registro de 8 bits se usa como salida para refrescar memorias externas y también como un generador de números pseudoaleatorios.
- W y Z. Estos dos registros son invisibles al programador y se utilizan para almacenar resultados temporales al ejecutar ciertas instrucciones del procesador.

Por su parte, el registro F se divide en los siguientes flags:

- Bit 7: Flag S. Es el flag que indica el signo, es decir, una copia del bit más significativo de la última operación llevada a cabo en la ALU.
- Bit 6: Flag Z. Este flag indica si el resultado de la última operación es cero.
- Bit 5: Flag 5. De este bit, no documentado por el fabricante, en algunos documentos se dice que tiene un valor aleatorio, pero en otros se especifica que guarda una copia del bit 5 del resultado de la última operación.
- Bit 4: Flag H. Guarda el acarreo del bit 3 al 4 de la operación, útil para el redondeo en operaciones en BCD.

- Bit 3: Flag 3. Este bit, tampoco documentado por el fabricante, actúa de forma análoga al Bit 5, pero guardando una copia del bit 3 del resultado de la última operación.
- Bit 2: Flag P/V. Dependiendo de la operación, en este bit se muestra si el resultado tiene paridad par (existe un número par de unos en el registro) o bien hubo desbordamiento.
- Bit 1: Flag N. Se activará si la última operación fue una resta.
- Bit 0: Flag C. Es el bit de acarreo, se activará si el resultado de la operación no entra en el registro.

2. Interrupciones en el Z80.

Una interrupción es una señal por la cual se interrumpe momentáneamente el secuenciamiento del programa, se trata dicha interrupción mediante una rutina y después se vuelve al punto donde se interrumpió.

Existen dos tipos de interrupciones: enmascarables (INT) y no enmascarables (NMI). La diferencia entre ambas es que las primeras pueden ignorarse si así se configura el computador y las segundas hay que tratarlas obligatoriamente.

En el Z80 existen dos biestables (dispositivos que almacenan un bit) llamados IFF1 e IFF2. El primero se usa para indicar si las interrupciones enmascarables están habilitadas (IFF1=1) o inhibidas (IFF1=0) y el segundo, como se explicará posteriormente, servirá como respaldo del primero.

Para habilitar las INT se usa la instrucción EI que pone a 1 los biestables IFF1 y IFF2, mientras que para inhibirlas se emplea la instrucción DI que los pone a 0.

Cuando se produce una NMI el procesador sigue los siguientes pasos:

1. Salvaguarda el contador de programa (PC) en la pila.
2. Guarda el valor de IFF1 en IFF2 y pone IFF1 = 0 (inhibe INT).
3. Pone PC = 0x0066 (Salta a la dirección 0x0066).

Una vez realizado este proceso, en la dirección 0x0066 estará la rutina correspondiente a su tratamiento. Cabe destacar que no se puede interrumpir a una NMI con una INT porque la primera pone IFF1 a 0.

Finalmente, cuando acabe la rutina de tratamiento de interrupción se ejecutará la instrucción RETN que copia IFF2 en IFF1 (recupera el valor guardado) y restaura el PC desde la pila para continuar la ejecución donde se quedó.

Hay tres modos diferentes en que puede estar el procesador para atender a las INT, estos modos llamados 0, 1 y 2 se establecen con las instrucciones IM0, IM1 e IM2.

El modo 0 es similar al único modo de operación del 8080 de Intel. El dispositivo que genera la INT pondrá el código de operación de una instrucción (normalmente una llamada a subrutina) en el registro de instrucciones y a partir de ahí seguirá la ejecución.

Por su parte, el modo 1 es análogo al tratamiento de las NMI, solo que ahora se salta a la dirección 0x0038 y es enmascarable.

Finalmente, en el modo 2 se usan interrupciones vectorizadas. El dispositivo pone en el bus de datos un byte. Este byte se tratará como la parte baja de una dirección de 16 bits y el valor del registro I como la parte alta de dicha dirección. Por último, se pondrá a 0 el bit menos significativo para que sean direcciones pares.

En esta dirección se encontrará la dirección de la rutina de tratamiento de interrupción adecuada a la que se saltará (previa salvaguarda del PC en la pila).

Cabe destacar que las interrupciones no enmascarables pueden ser anidadas (una interrupción interrumpe a otra), pero no pueden saltar si se está tratando otra no enmascarable.

Interrupciones en la Master System

En la Master System el único dispositivo que provoca interrupciones es el procesador gráfico (VDP). Aunque se tratará con más profundidad en su capítulo correspondiente, sería conveniente adelantar que, en concreto, genera una interrupción en dos momentos diferentes: al acabar de pintar un fotograma y al acabar de pintar una línea.

Por otro lado, según el manual de referencia de la propia SEGA, la Master System solo atiende al modo de interrupción 1, así que a menos que los programadores se salten las recomendaciones en sus programas y establezcan otro modo sería suficiente con implementar éste.

Respecto a las NMI, su línea está directamente conectada con el botón de Pausa de la consola, así que su pulsación es el único momento en que se generará dicho tipo de interrupción.

3. Formato de instrucciones del Z80.

Todos los programas escritos para un microprocesador (en este caso los contenidos de las ROMs de los cartuchos de Master System) no son más que una sucesión de instrucciones comprensibles por dicho procesador.

Cada instrucción específica afectará de diferente manera a ciertos elementos del procesador (registros principalmente) o a dispositivos conectados a él (memorias y otros dispositivos de entrada/salida).

De esta manera se pueden dividir las instrucciones en varios grupos como serían las de carga (mueven información entre registros o registros y memoria), aritméticas y lógicas (operan con los datos), de salto y bifurcaciones, etc.

Las instrucciones están formadas siempre por un código de operación (un prefijo que indica de qué instrucción se trata) y cero o más parámetros necesarios para la instrucción en cuestión.

En el caso del Z80, un procesador CISC, los códigos de operación de su juego de instrucciones siempre ocupan un byte, por lo tanto existen 256 códigos de operación diferentes que resultarían en unas teóricas 256 instrucciones diferentes. Sin embargo, esto último no es cierto; en este procesador existen ciertos códigos de operación que se usan como un prefijo, de modo que su siguiente parámetro (otro byte) se usará como un subcódigo de operación que añadirá otras 256 instrucciones diferentes.

En el Z80 existen 4 de estos prefijos: CB, DD, ED y FD, pero aún hay más, hay otros dos prefijos de segundo nivel que abren otras 256 posibles instrucciones cada uno: DDCB y FDCB. En definitiva, gracias a estos prefijos se dispone de 1.268 instrucciones (algunos códigos de operación no se utilizan).

Cada una de estas instrucciones tiene una correspondencia en ensamblador (un código mnemónico) para facilitar la programación por parte del usuario, así la instrucción de CO (código de operación) 0x00 se corresponde con el mnemónico *NOP*, 0xD1 con *POP DE* o 0xDD94 con *SUB IXh*.

El procesador a la hora de ejecutar un programa no hace más que ir leyendo de las posiciones de memoria apuntadas por el registro contador de programa y ejecutando la instrucción correspondiente al código de operación allí encontrado.

A la hora de programar el núcleo del Z80 para el emulador la tarea más larga y tediosa por repetitiva es la de implementar estas 1.268 instrucciones, pues si bien muchas de ellas son iguales solo que cambiando los parámetros (mover cada uno de los distintos registros al acumulador, por ejemplo), aun así se quedan en unas 150 instrucciones totalmente diferentes unas de otras.

En este emulador, para obtener una implementación más eficiente, se ha optado por crear una tabla con todas las instrucciones (las 1.268) y se saltará a una u otra tras analizar el CO leído.

4. Grupos de instrucciones.

En el Z80 se pueden distinguir once grupos de instrucciones, en concreto los siguientes:

1. Carga de 8 bits. Las instrucciones de este grupo se centran en realizar la transferencia de datos de 8 bits entre registros de la CPU y posiciones de memoria.
2. Carga de 16 bits. Las instrucciones de este grupo se utilizan para transferir datos de 16 bits entre pares de registros de 8 bits, registros de 16 bits y palabras de memoria agrupadas como pares. También se engloban aquí las instrucciones de manejo de la pila (*PUSH* y *POP*).
3. Búsqueda, intercambio y transferencia de bloques. En este variado grupo se engloban las instrucciones usadas para el intercambio de datos entre registros de distintos bancos o entre memoria y registros. También se agrupan las instrucciones de transferencias de bloques de memoria.
4. Aritmético-lógicas de 8 bits. En este grupo se sitúan las instrucciones usadas para realizar operaciones aritméticas o lógicas con datos de 8 bits.
5. Aritméticas de propósito general y control de CPU. Aquí se agrupan instrucciones aritméticas sin parámetros que afectan al acumulador y otras instrucciones de control de la CPU como puede ser la de reinicio, parada o control de interrupciones.
6. Aritméticas de 16 bits. Las instrucciones aritméticas que operan con registros de 16 bits se engloban en este grupo.
7. Rotación y desplazamiento. En este apartado se sitúan las instrucciones relacionadas con rotaciones y desplazamientos de los bits que contienen los registros.
8. Establecimiento, borrado y comprobación a nivel de bits. Aquí se engloban las instrucciones usadas para cambiar el valor individual de cada uno de los bits de los registros a 0 o 1. También aparecen instrucciones para comprobar el valor de cada bit por separado.
9. De saltos. En este grupo se sitúan las instrucciones usadas para provocar saltos relativos o absolutos en el secuenciamiento del programa mediante cargas en el contador de programa.

10. De llamadas y retorno. Como su nombre indica, las instrucciones que provoquen llamadas a subrutinas y el retorno desde éstas se situarán en este grupo. También se sitúa aquí la instrucción de retorno de interrupción.
11. Entrada y salida. Aquí se agruparán las instrucciones de transferencia de información entre registros y dispositivos periféricos a través de los puertos de entrada/salida de la CPU.

Para ver las instrucciones específicas que componen estos grupos y cada una de éstas en detalle se ha incluido el Apéndice D (Instrucciones detalladas del Z80) que puede encontrarse en este mismo documento.

5. El ciclo de instrucción.

Los procesadores son máquinas secuenciales, es decir, su funcionamiento se basa en ejecutar una tras otra las instrucciones contenidas en memoria.

El ciclo de instrucción podemos dividirlo en tres partes:

1. Tomar la siguiente instrucción (*fetch*).
2. Decodificar la instrucción.
3. Ejecutar la instrucción.

El procesador siempre tiene su registro contador de programa apuntando a la dirección de memoria de la siguiente instrucción a ejecutar. Durante la fase de *fetch* el procesador extrae la instrucción de la memoria apuntada por el PC volcando el contenido de este registro al bus de direcciones y pidiendo una operación de lectura a la memoria. Simultáneamente a esta petición, el PC se incrementará en una unidad para apuntar a la siguiente instrucción o bien al primer operando de la instrucción extraída si ésta lo tuviera.

Cuando la memoria está preparada para atender la petición, vuelca el contenido de la dirección pedida al bus de datos de donde el procesador recoge el código de operación y lo coloca en un registro especial llamado registro de instrucción (IR).

Una vez tiene el código de operación alojado en el IR, el procesador decodifica éste para saber de qué instrucción se trata y obtiene los parámetros de la memoria (si los tuviera), incrementando el PC en una unidad por cada parámetro extraído.

Llegado a este paso, el procesador ya sabe de qué instrucción se trata y los parámetros que necesita, luego simplemente la ejecuta de la forma apropiada.

Al finalizar este paso finaliza una iteración del ciclo de instrucción, se vuelve al *fetch* de la siguiente instrucción y así sucesivamente.

6. Inicio y reinicio de la CPU.

Cada vez que se inicia la CPU o bien después de un reinicio software, los registros y biestables de ésta deben tomar unos valores determinados para que comience correctamente el secuenciamiento del programa.

En concreto, todos los registros tomarán el valor 0xFF a excepción del PC el cual se pondrá a 0x0000 para que el programa existente a partir de dicha dirección de memoria tome el control. Así mismo, los registros especiales I y R tomarán también el valor 0x00.

Por su parte, las interrupciones se inicializarán con los biestables IFF1 e IFF2 a 0 (interrupciones inhibidas) y en modo de interrupción 0.

7. Implementación.

El procesador Z80 se ha implementado como la clase mostrada en la figura 5.

Cabe hacer especial hincapié en la estructura contextoZ80 mostrada en la figura 6. En este tipo definido se guarda toda la información relativa al estado interno del Z80, almacenando el contenido de cada uno de los registros, el estado de las interrupciones o información relativa a los ciclos de reloj ejecutados. Apoyándose en esta estructura será posible salvar y cargar instantáneas de su estado tan solo guardando estos datos a disco, lo que será utilizado para los *savestates* de la consola.

Comenzando con los temas tratados, la implementación de las interrupciones, como se puede ver en la figura 7, es bastante trivial, tan solo cabe destacar la actualización del registro R en la cual se le suma una unidad cada vez que se atiende una interrupción, pero manteniendo siempre inalterable el bit más significativo del mismo.

Respecto al ciclo de instrucción, hay múltiples formas de implementarlo, una por cada programador, en este caso se ha optado por una aproximación entre lo didáctico y lo eficiente.

Este emulador se basa en una implementación orientada a objetos, en este caso habrá un objeto de la clase Z80 y otro objeto de la clase Memoria relacionados entre sí de forma que desde el Z80 haya visibilidad hacia la Memoria, pero no al revés.

Por otro lado, habrá un programa principal que tiene instancias de estas dos clases y será el encargado de controlar la emulación. Desde este programa principal se pedirá

al Z80 que ejecute x ciclos de reloj, llevando a cabo las instrucciones que dé tiempo en esos ciclos (cada instrucción consume unos determinados ciclos).

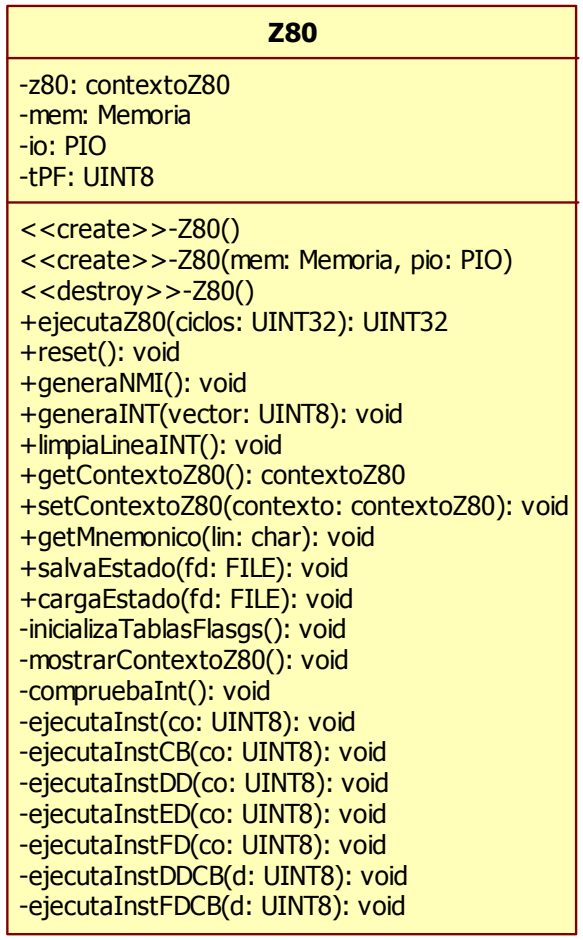


Fig. 5. Clase Z80

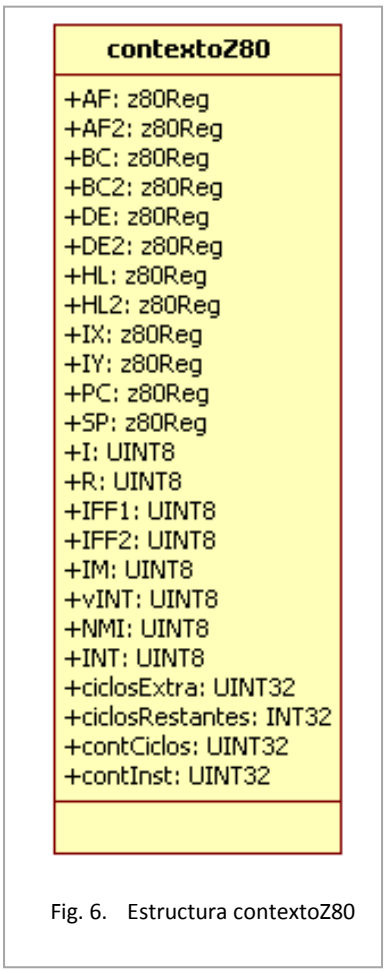


Fig. 6. Estructura contextoZ80

```
void Z80::compruebaInt()
{
  if (_NMI)
    {
      _R = (_R & SF) | (_R+1 & 0x7F);
      PUSH(_PC);
      _IFF2 = _IFF1;
      _IFF1 = 0;
      _NMI = 0;
      _PC = 0x66;
    }
  else if (_IFF1 && _INT)
    {
      _R = (_R & SF) | (_R+1 & 0x7F);
      _INT = 0;
      _IFF1 = _IFF2 = 0;
      switch (_IM)
      {
        case 0:
          ejecutaInst(_VINT);
          break;
        case 1:
          PUSH(_PC);
          _PC = 0x38;
          break;
        case 2:
          PUSH(_PC);
          _PC = (_I<<8) | _VINT;
          break;
      }
    }
}
```

Fig. 7. Rutina de tratamiento de interrupción

Como se puede ver en la figura 8, se le pasa la cantidad de ciclos a ejecutar y devolverá los que realmente se han ejecutado.

Las instrucciones se ejecutan íntegramente (no se puede ejecutar media instrucción o tres cuartos de instrucción), así que es muy posible que se ejecuten más ciclos de los que se han pedido. Por ejemplo, si se pidiera ejecutar 5 ciclos y en el programa se tuviera una instrucción de 3 y otra de 6 ciclos realmente ejecutará 9. Por este motivo se guarda en *ciclosRestantes* los ciclos a ejecutar sumados a los que quedaban de la anterior ejecución (éstos serán un número negativo) de modo que si en la anterior ejecución se ejecutaron 4 ciclos de más y en ésta se le pidiera ejecutar 7, realmente se intente ejecutar solo la diferencia, es decir, 3 ciclos.

```
UINT32 Z80::ejecutaZ80(UINT32 ciclos)
{
  z80.ciclosRestantes += ciclos;
  UINT32 ciclosEjecutados = z80.ciclosRestantes;
  do
    {
      compruebaInt();
      ejecutaInst(mem->readMem(_PC++));
      z80.contInst++;
    } while (z80.ciclosRestantes > 0);
  ciclosEjecutados -= z80.ciclosRestantes;
  z80.contCiclos += ciclosEjecutados;
  return ciclosEjecutados;
}
```

Fig. 8. Ciclo de instrucción

La fase de *fetch* se resuelve en `mem->readMem(_PC++)`, pues se toma la instrucción de la dirección de memoria apuntada por el PC y se incrementa éste.

Se detendrá la ejecución de instrucciones cuando *ciclosRestantes* sea igual o menor que cero (como se verá posteriormente esta variable se decrementa en cada ejecución de instrucción).

La rutina *ejecutaInst* es la que se encarga de decodificar y ejecutar la instrucción. El modo más eficiente de hacerlo hubiera sido creando una función para cada código de operación y luego una tabla con punteros a estas funciones, pero la implementación realizada ha sido otra.

Al hacerlo en C++, cada función de la clase debe estar declarada en la parte pública o privada de la misma. Meter ahí 1268 funciones (una por cada código de operación) es algo desproporcionado, por otro lado, aun agrupándolas en funcionalidades similares saldrían más de 70, lo cual sigue siendo excesivo. Debido a esto se ha tomado la determinación de hacer un *switch* con todos los códigos de operación (que el compilador convertirá en una tabla para optimizar los saltos a la selección apropiada) e introducir en cada uno de éstos el código de las instrucciones.

Escribir seguido todo este código aparte de tedioso es muy poco elegante, así que ahí entran en juego las macros del preprocesador de C.

La función *ejecutaInst* quedaría algo así:

```
void Z80::ejecutaInst(UINT8 co)
{
  z80.ciclosRestantes -= cc[co];
  switch (co)
  {
    case 0x00: NOP; break;
    case 0x01: LD16_R_I(_BC); break;
    case 0x02: LD_D_R(_BC,_A); break;
    .
    .
    .
    .
  }
}
```

Fig. 9. Función *ejecutaInst*

Como se puede apreciar, resta una cantidad de ciclos específica de cada instrucción (contenido su valor en la tabla *cc* a la que se accede por el código de operación) y, dependiendo del código de operación, ejecutará una macro. En esta rutina se realizan las fases de decodificación y ejecución.

A continuación se detallan un par de estas macros a modo de muestra:

```
/***********************************************
 * LD (dir), r - 8 bits
 ***********************************************/
#define LD_D_R(dir,origen)                      \
{                                               \
  WM(dir, origen);                              \
}

/***********************************************
 * LD dd, nn - 16 bits
 ***********************************************/
#define LD16_R_I(destino)                       \
{                                               \
  destino = RM(_PC++) | (RM(_PC++)<<8);         \
}
```

Fig. 10. Macros de instrucciones de ejemplo

Por su parte, la implementación del reinicio de la CPU, una vez expuesta la información de su apartado específico, es bastante trivial. Como se puede ver en la figura siguiente se limita a establecer los valores por defecto de los distintos registros almacenados en la estructura contextoZ80 y reiniciar los contadores de ciclos ejecutados.

```
void Z80::reset()
{
  _AFD = _AFD2 = 0xFFFF;
  _BCD = _BCD2 = 0xFFFF;
  _DED = _DED2 = 0xFFFF;
  _HLD = _HLD2 = 0xFFFF;
  _IX = _IY = _SP = 0xFFFF;
  _PC = 0;
  _I = _R = _IFF1 = _IFF2 = _IM = 0;
  _NMI = _INT = _VINT = 0;
  z80.contCiclos = z80.contInst = 0;
  z80.ciclosRestantes = 0;
}
```

Fig. 11. Reset

Capítulo 4. El sistema de entrada/salida y memoria.

1. Espacios direccionables independientes.

Como se comentó en el capítulo anterior, el Z80 tiene un bus de direcciones de 16 bits y, por tanto, puede direccionar hasta 65.536 palabras de un byte de longitud.

Gracias a su arquitectura de espacios direccionables independientes, el Z80 es capaz de seleccionar en un momento dado, mediante una señal, si comunicarse con dispositivos de entrada/salida o memoria. Esto permite el uso de todo el espacio direccionable para cada uno de los dos subsistemas.

2. Mapa de memoria.

Tal y como se puede ver en la figura 12, la Master System cuenta con 8 Kbytes de memoria RAM que se encuentran mapeados a partir de la dirección 0xC000, pero también son accesibles desde la dirección 0xE000 (es lo que se conoce como RAM duplicada).

Esta duplicación del acceso al mismo módulo de memoria tiene un sentido. Las cuatro últimas direcciones de memoria (0xFFFC-0xFFFF) tienen un significado especial, pues en ellas hay mapeados unos registros de solo escritura llamados registros de marco. Estos registros no pueden leerse por sí mismos, pero sí es posible leer de las direcciones equivalentes en la otra zona de RAM (0xDFFC-0xDFFF) para obtener su valor.

Cuando se inicia la Master System en la zona inferior de su memoria se encuentra una pequeña ROM (de 8 Kbytes) que contiene la BIOS. Ésta se encargará de inicializar ciertos registros y copiarse a sí misma a la memoria RAM. Si encuentra un cartucho (con su correspondiente módulo de memoria ROM) introducido en la ranura del mismo, le cede el control, mapeando la ROM del cartucho a partir de la dirección 0x0000.

Como se puede deducir de lo anteriormente explicado, a priori, para la ROM del programa a cargar solo se dispone de 48 Kbytes de memoria direccionables (desde la dirección 0x0000 a 0xBFFF). Esto físicamente es así y de este modo se cargan los programas de hasta 32 Kbytes de memoria, pero ya se comentó que hay programas que pueden llegar al Mbyte de memoria, ahí entran en juego los mapeadores de memoria.

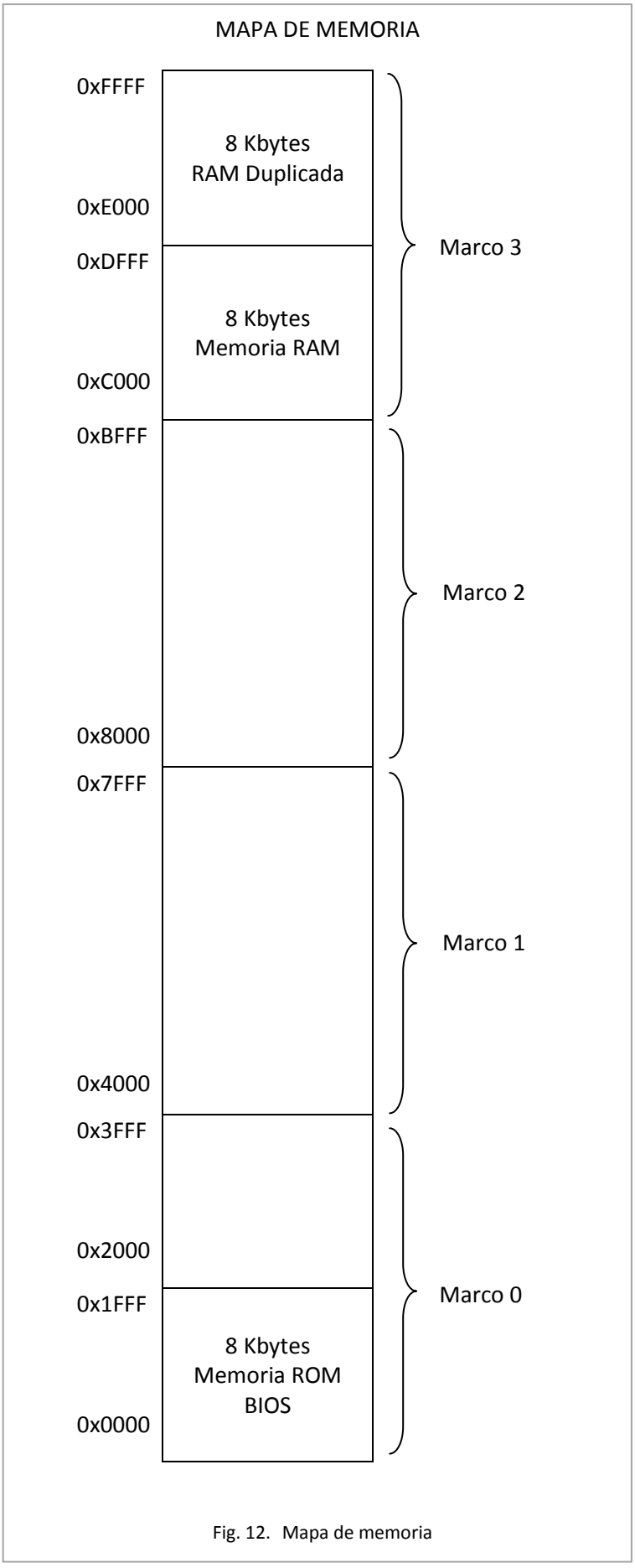


Fig. 12. Mapa de memoria

3. Los mapeadores de memoria.

Si bien los programas más básicos incorporaban tan solo un módulo de 32 Kbytes de ROM en su cartucho, los programas de mayor capacidad necesitan de un hardware adicional conocido como mapeador de memoria.

Al usar los mapeadores de memoria se dividen los 64 Kbytes direccionables por el procesador en 4 marcos de 16 Kbytes cada uno numerados de 0 (el correspondiente a las direcciones más bajas) a 3 (las direcciones más altas).

Así mismo, la ROM del programa en cuestión también se dividirá en tantos bancos de 16 Kbytes como sea necesario para cubrir todo su tamaño.

Mediante los registros de marco vistos anteriormente, el programa podrá seleccionar qué banco de la ROM mapear en cada momento en qué marco de memoria (del 0 al 2, excluyendo el banco 3 ocupado por la RAM), posibilitando de este modo el acceso a cualquier tamaño de memoria.

Cabe destacar que las primeras 1024 direcciones de memoria (0x0000-0x0400) no son expulsables, es decir, siempre corresponden a los primeros 1024 bytes de la ROM del programa en cuestión. Esto es debido a que los vectores de interrupción se encuentran en esa zona de memoria y de este modo se evita tener que copiar el código relativo a ellos en cada banco de memoria de la ROM para que se mantengan siempre constantes.

En la figura 13 se pueden apreciar dos casos diferenciados. El primero es un programa de 32 Kbytes donde se observa que se mapea directamente en los 32 Kbytes inferiores de memoria. En la segunda imagen se ve un programa de 128 Kbytes en el que, por decisión del programador, se mantienen fijos los dos marcos inferiores con los bancos 0 y 1 y se usa el tercero para mapear cualquiera de los otros 6 bancos.

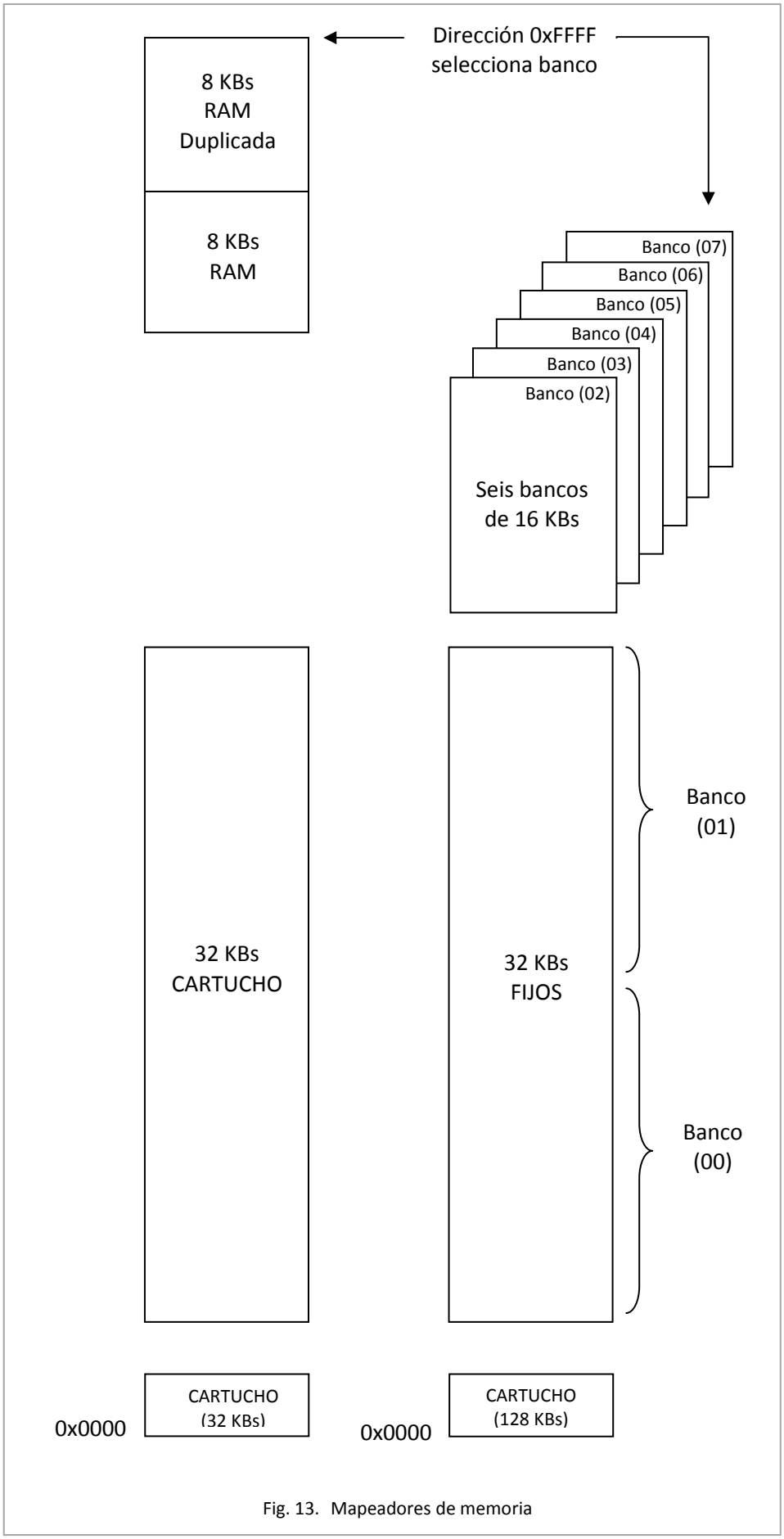


Fig. 13. Mapeadores de memoria

4. Los registros de marco.

En la siguiente figura se detallan los valores que pueden tomar los registros de marco.

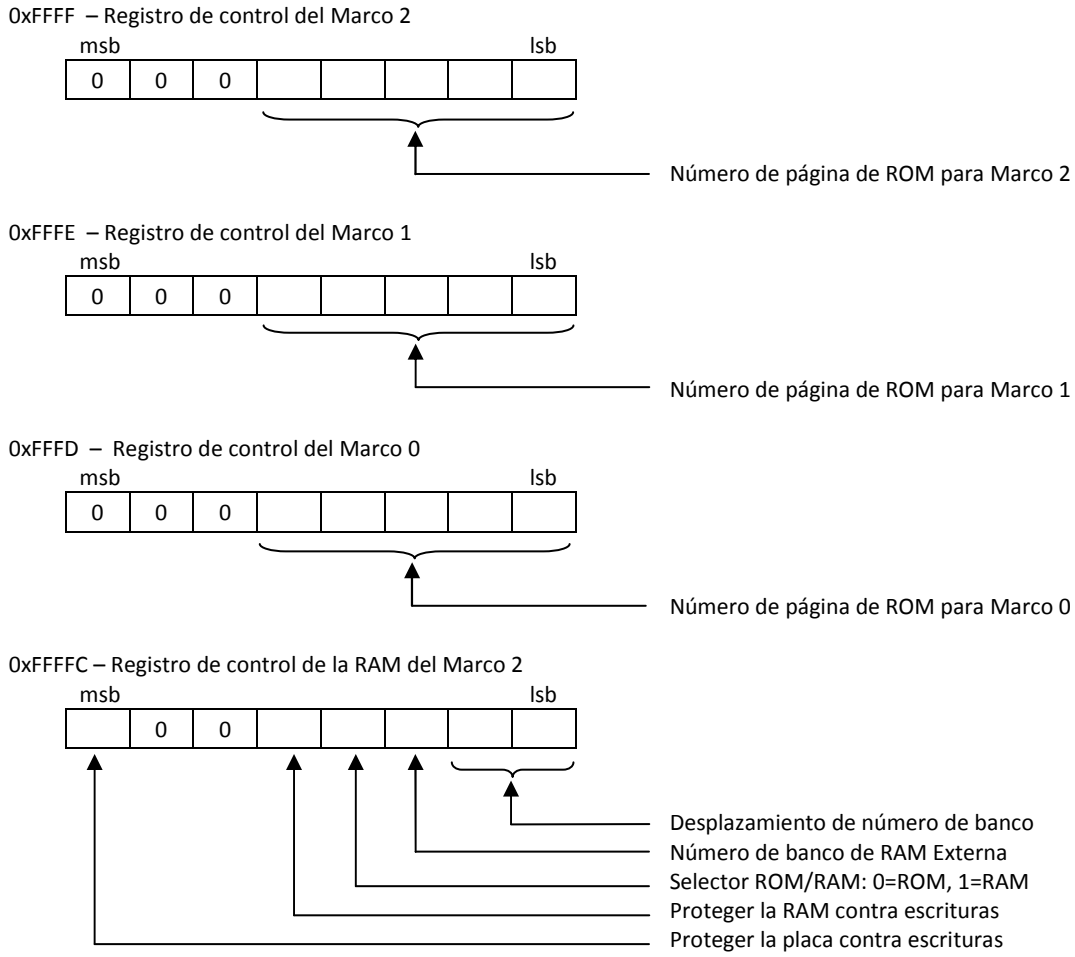


Fig. 14. Registros de marco

Al arrancar la máquina la BIOS establece los siguientes valores:

0xFFFC = 0 0xFFFD = 0 0xFFFE = 1 0xFFFF = 2

5. Los puertos de entrada/salida.

A pesar del enorme espacio de entrada/salida proporcionado por el Z80, la Master System solo hace uso de una pequeña cantidad de estos puertos de entrada/salida, donde tiene conectados una serie de dispositivos externos al microprocesador.

En concreto se utilizan los puertos que aparecen en la siguiente figura:

Puerto	Dirección	Función
0x3E	Salida	Activa la memoria.
0x3F	Salida	Registro de control del joystick.
0x7E	Entrada	Contador vertical del VDP.
0x7F	Entrada	Contador horizontal del VDP.
	Salida	Registro de control del PSG.
0xBE	Entrada/Salida	Registro de datos del VDP.
0xBF	Entrada/Salida	Registro de control/estado del VDP.
0xC0	Entrada	Registro de estado del joystick.
0xC1	Entrada	Registro de estado del joystick.
0xDC	Entrada	Registro de estado del joystick.
0xDD	Entrada	Registro de estado del joystick.

Fig. 15. Puertos de entrada/salida

El puerto 0x3E se utiliza para funciones de inicialización de la memoria, pero a la hora de programar un emulador no será necesario, pues se puede suponer que la memoria siempre estará inicializada correctamente.

Los puertos 0x3F, 0xC0, 0xC1, 0xDC y 0xDD son relativos a los dispositivos controladores.

Los puertos 0x7E, 0x7F (cuando se lee de él), 0xBE y 0xBF se emplean para comunicarse con el procesador de video.

Finalmente, el puerto 0x7F (cuando se escribe en él) se utiliza para programar el generador de sonido.

El detalle de las acciones realizadas al leer o escribir en estos puertos se especificará en los capítulos correspondientes a los dispositivos implicados.

6. Implementación.

Para la realización del emulador se han implementado las clases que se pueden ver en las siguientes figuras:

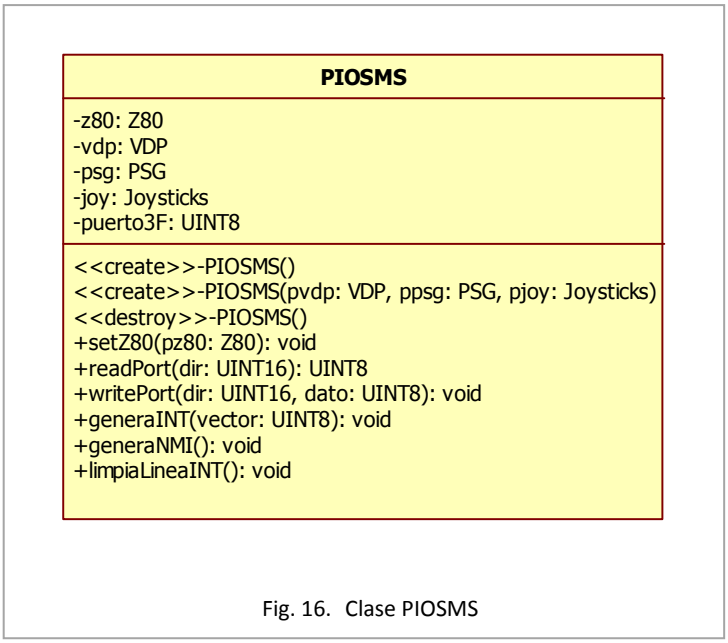


Fig. 16. Clase PIOSMS

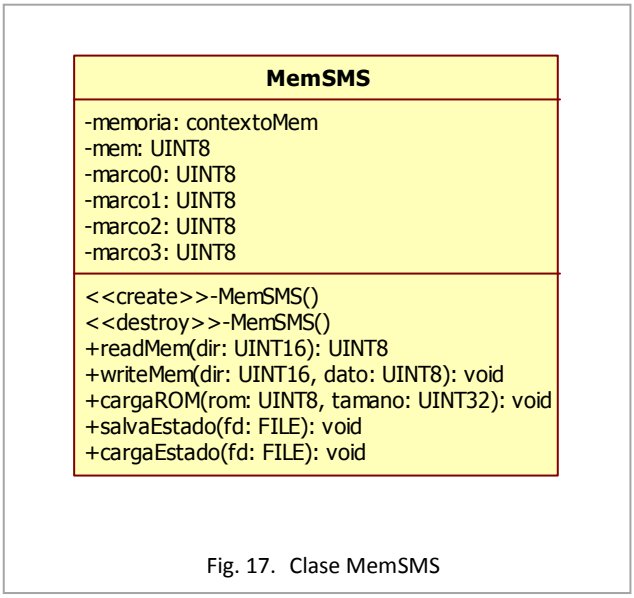


Fig. 17. Clase MemSMS

Como se puede ver, ambas clases son muy similares, destacando en las dos sus operaciones de lectura y escritura.

La clase correspondiente a la entrada/salida (PIOSMS) posee además métodos para generar INT y NMI así como para “limpiar” la línea de petición de INT del procesador. Estos métodos se usarán como “puente” entre el procesador y los dispositivos conectados a éste, propagando las señales de uno a otro.

La clase correspondiente a la memoria (MemSMS) posee también métodos para cargar un programa (*cargaROM*) en el espacio de memoria así como para salvar el estado de la memoria y su contenido que será necesario a la hora de cargar o salvar el estado de la máquina a emular.

De este modo las instancias de la clase MemSMS servirán para proporcionar instrucciones a la instancia de la clase Z80 e intercambiar datos con ella, mientras que las instancias de la clase PIOSMS se utilizarán para comunicar la instancia de la clase Z80 con cada una de las instancias de las clases correspondientes a los dispositivos conectados a ésta (VDP, PSG y Joysticks).

Esta última clase “puente “ (PIOSMS) es necesaria puesto que el procesador podría tener conectados múltiples dispositivos, pero es el controlador de entrada/salida quien lleva las señales escritas en cada puerto al dispositivo correspondiente.

Con la información proporcionada en este capítulo la implementación de estas clases es bastante trivial, tan solo cabría comentar que el acceso a los bancos de memoria de la ROM se ha realizado mediante punteros base de forma que cada marco del sistema de memoria pueda apuntar a cualquier banco tan solo cambiando la dirección contenida en el puntero.

Capítulo 5. El procesador de video (VDP).

1. Descripción general.

La Master System lleva incorporado un procesador de video dedicado conocido como VDP (las siglas del término inglés *Video Display Processor*), cuya función es liberar al Z80 de computar gran parte de la carga gráfica.

Este procesador está basado en un TMS9918A de Texas Instruments pero con algunos aspectos mejorados.

Si bien el VDP soporta 16 modos de video distintos, tan solo uno de ellos, el modo 4, está documentado por SEGA, siendo su resolución de 256x192 píxeles.

El VDP cuenta con dos pequeñas memorias de tipo RAM internas al propio chip.

La primera es la llamada CRAM (Color RAM) que se encarga de almacenar las paletas de colores que se pueden usar en pantalla y tiene un tamaño de 32 bytes, lo que resulta en 32 colores RGB distintos de 6 bits cada uno (2 bits para almacenar cada tono: rojo, verde y azul) de un total de 64.

La segunda memoria es la VRAM (Video RAM) y tiene un tamaño de 16 Kbytes. Esta memoria se divide en tres zonas:

1. Mapa de Pantalla (1.792 bytes): es una zona dedicada a almacenar un mapa con la información de los *tiles* que forman el fondo (32x24 *tiles* de 8x8 píxeles).
2. Tabla de Atributos (256 bytes): una zona dedicada a almacenar una tabla con los atributos de un máximo de 64 *sprites* simultáneos.
3. Generador de Caracteres (14.336 bytes): es la zona de memoria donde se almacenan los caracteres, es decir, pequeños gráficos de 8x8 píxeles de los que están formados los *tiles* y *sprites*.

Por otro lado, el VDP cuenta con 12 registros internos de 8 bits que sirven para determinar su comportamiento.

Cabe mencionar que el Z80 se comunica con el VDP a través de dos puertos de entrada/salida, concretamente los 0xBF (comandos) y 0xBE (datos).

2. Caracteres, *tiles* y *sprites.*

En este punto habría que aclarar tres elementos esenciales en el procesador gráfico de la Master System: los caracteres, *tiles* y *sprites*.

Los caracteres son pequeños gráficos de 8x8 píxeles que se usarán como base para componer cualquier elemento que aparezca en pantalla (*tiles* y *sprites*). Es decir, los caracteres son la mínima unidad representable, no es posible dibujar un píxel solitario sino que en su lugar habría que dibujar un carácter con un píxel de un color y los otros 63 de color transparente.

Los *tiles*, como su nombre indica, son baldosas con las que se forma el fondo de la imagen. Su particularidad es que cada *tile* siempre tiene una posición fija en la pantalla, en cada *tile* se dibujará un carácter y por tanto habrá 32x24 *tiles* visibles.

Los *sprites* por su parte son gráficos (también se corresponden 1:1 con un carácter) que tienen la particularidad de poder situarse en cualquier posición de la pantalla.

En la práctica, los *tiles* se usarán para dibujar los escenarios de los juegos (que permanecen fijos) y los *sprites* para dibujar los personajes, enemigos u otros elementos móviles.

3. Las paletas.

El VDP de la Master System trabaja en un modo de video paletizado. Esta técnica se utiliza para ahorrar memoria de video, representando el color de cada píxel como un índice a una tabla (la paleta de colores) en lugar del valor del propio color.

En el caso concreto de la Master System el color de cada píxel viene representado por un número de 4 bits que actúa como un índice sobre la paleta, pudiendo seleccionar uno de 16 colores. Esta paleta a su vez está formada por 32 entradas de 1 byte, cada una representando a un color de 6 bits en RGB (2 bits para cada color, de la forma xxBBGGRR).

Con este sistema paletizado se puede hacer uso de 64 colores diferentes en pantalla con solo 4 bits de información por cada píxel.

Como es por todos sabido, con 4 bits solo se puede direccionar 16 elementos, pero la paleta tiene 32 entradas. Efectivamente, hay que establecer una división entre los 16 primeros y los 16 segundos colores. Los *sprite*s solo pueden acceder a estos últimos mientras que los *tiles* pueden acceder a cualquiera de los dos bancos por medio de un bit adicional que llevan en su descripción.

4. El Generador de Caracteres.

Como ya se ha comentado, los caracteres son la mínima unidad representable y ocupan 8x8 píxeles.

Cada carácter está representado por 32 bytes de la VRAM del VDP, luego teniendo 16 Kbytes de RAM es trivial deducir que se pueden representar 512 caracteres en dicha memoria. Aunque, como se verá más adelante, hay zonas de esta memoria ocupadas por otras unidades de información.

De los 32 bytes de cada carácter, cada grupo de 4 se usa para representar una fila de las 8 que lo componen. El color se obtiene de los 4 planos de bits formados por estos 4 bytes.

Por ejemplo, el primer carácter (ocupa las direcciones 0 a 31 de la VRAM) podría estar formado por los bytes: FF, 00, FF, 00, 00, 00, C0, C0, 00, 00, C0, C0, 00, 00, C0, C0, 00, 00, C0, C0, 00, 00, C0, C0, 00, 00, C0, C0, 00, 00, C0, C0, 00, 00, C0, C0, 00, FF, 00 y 00.

En la figura 19 se puede ver cómo quedarían desglosados sus bits.

Como se puede observar, cada cuatro bytes forman una fila y el color asociado se obtiene leyendo de "abajo a arriba", es decir, del byte 3 al 0, del 7 al 4, etc.

En este ejemplo solo se usan 4 colores, en concreto los correspondientes a los índices de la paleta número 0, 2, 5 y 12. Como se verá posteriormente, estos colores pueden ser cualquiera de los soportados siempre que estén en la posición indicada dentro de la paleta.

Suponiendo que los índices se correspondieran con los siguientes colores (podría ser cualquier color, esto es un supuesto):

0 = Blanco, 2 = Rojo, 5 = Azul, 12 = Verde.

Quedaría una imagen como la de la siguiente figura:

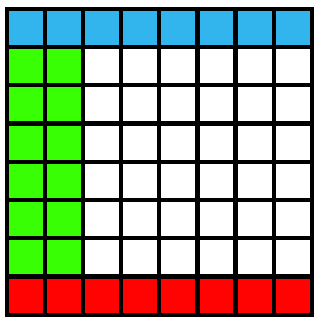


Fig. 18. Representación gráfica de un carácter

```
Byte b7 b6 b5 b4 b3 b2 b1 b0
0    1  1  1  1  1  1  1  1   c0  PRIMERA FILA DE PÍXELES
1    0  0  0  0  0  0  0  0   c1    '0101' = color #5
2    1  1  1  1  1  1  1  1   c2
3    0  0  0  0  0  0  0  0   c3

4    0  0  0  0  0  0  0  0   c0  SEGUNDA FILA DE PÍXELES
5    0  0  0  0  0  0  0  0   c1    '1100' = color #12,
6    1  1  0  0  0  0  0  0   c2    '0000' = color #0.
7    1  1  0  0  0  0  0  0   c3

8    0  0  0  0  0  0  0  0   c0  TERCERA FILA DE PÍXELES
9    0  0  0  0  0  0  0  0   c1
10   1  1  0  0  0  0  0  0   c2
11   1  1  0  0  0  0  0  0   c3

12   0  0  0  0  0  0  0  0   c0  CUARTA FILA DE PÍXELES
13   0  0  0  0  0  0  0  0   c1
14   1  1  0  0  0  0  0  0   c2
15   1  1  0  0  0  0  0  0   c3

16   0  0  0  0  0  0  0  0   c0  QUINTA FILA DE PÍXELES
17   0  0  0  0  0  0  0  0   c1
18   1  1  0  0  0  0  0  0   c2
19   1  1  0  0  0  0  0  0   c3

20   0  0  0  0  0  0  0  0   c0  SEXTA FILA DE PÍXELES
21   0  0  0  0  0  0  0  0   c1
22   1  1  0  0  0  0  0  0   c2
23   1  1  0  0  0  0  0  0   c3

24   0  0  0  0  0  0  0  0   c0  SÉPTIMA FILA DE PÍXELES
25   0  0  0  0  0  0  0  0   c1
26   1  1  0  0  0  0  0  0   c2
27   1  1  0  0  0  0  0  0   c3

28   0  0  0  0  0  0  0  0   c0  OCTAVA FILA DE PÍXELES
29   1  1  1  1  1  1  1  1   c1    '0010' = color #2
30   0  0  0  0  0  0  0  0   c2
31   0  0  0  0  0  0  0  0   c3
```

Fig. 19. Carácter de ejemplo

A partir de pequeñas imágenes como ésta es como está formada toda el área visible que representará el VDP, es decir, los *sprites* y *tiles*.

5. El Mapa de Pantalla.

El Mapa de Pantalla es la zona de la memoria VRAM del VDP que se emplea para almacenar los datos que representarán el "escenario" (los *tiles*) sobre el que se moverán los *sprites*.

Como se puede ver en la figura 20, la pantalla está organizada como una matriz de 768 caracteres visibles: 32 horizontales por 24 verticales. Además de estos caracteres existen otras 4 filas de caracteres debajo de las 24 verticales, inicialmente no visibles, que se usarán para mostrar información adicional en el caso de desplazamientos verticales.

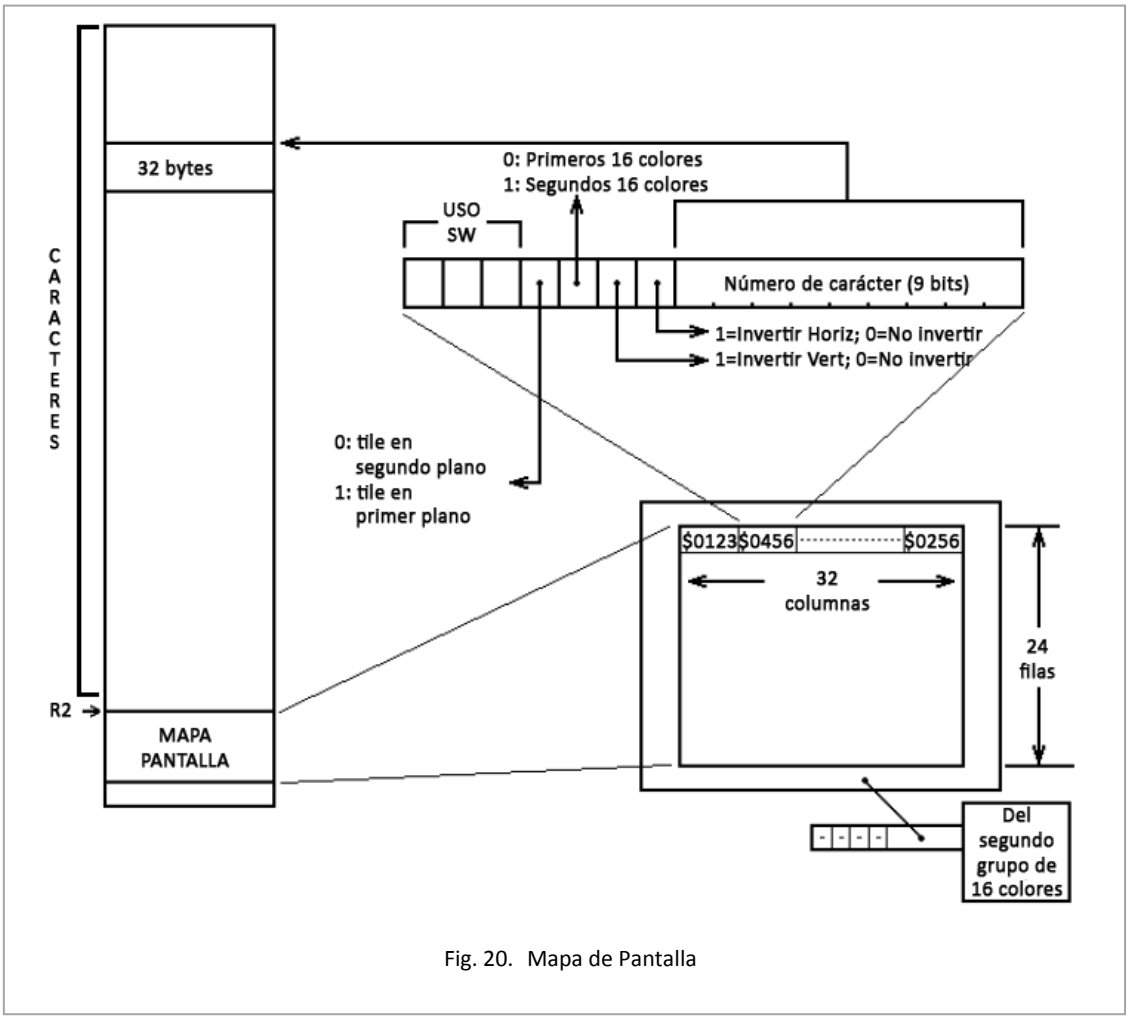


Fig. 20. Mapa de Pantalla

Recapitulando, los 2.048 bytes del Mapa de Pantalla definen la posición de 896 caracteres (768 visibles). Por cada uno de estos caracteres se usará una palabra de 16 bits que describe lo siguiente:

1. Cuál de los 512 caracteres disponibles mostrar en la posición del carácter (9 bits).
2. Si voltear o no el carácter horizontal o verticalmente (2 bits).
3. Cuál de los dos juegos de 16 colores de la paleta se usarán para el carácter (1 bit).
4. Si los *sprites* se superpondrán al *tile* o viceversa (1 bit).
5. Tres bits adicionales sin uso específico, útiles para usar como flags personalizables por parte de los programadores (3 bits).

El Mapa de Pantalla visible ocupa 1.536 bytes de la VRAM y las cuatro filas no visibles ocuparán otros 256 bytes, por tanto quedarían sin usar 256 bytes adicionales de memoria en la parte superior de los 2048 bytes. Estos 256 bytes normalmente se usarán para almacenar la Tabla de Atributos de *Sprites*.

6. La Tabla de Atributos de *Sprites*.

Los *sprites*, a diferencia de los *tiles*, son gráficos que pueden desplazarse por toda la pantalla, así que será necesario guardar la información de la posición de cada uno.

Al igual que los *tiles*, los *sprites* se corresponden con un carácter, pero mientras aquellos podían usar cualquiera de los dos grupos de 16 colores, los *sprites* solo pueden hacer uso del segundo grupo de la CRAM.

Otra consideración a tener en cuenta es que es recomendable que haya un máximo de ocho *sprites* simultáneos en la misma línea de la pantalla, de lo contrario se producen parpadeos por limitaciones del VDP.

Para almacenar la información relativa a los *sprites* se usará la Tabla de Atributos de *Sprites*, una pequeña zona de la VRAM del VDP de tan solo 256 bytes, pero suficientes para representar hasta 64 *sprites* simultáneos.

En la siguiente figura se puede apreciar la organización de la Tabla de Atributos de *Sprites*:

Dirección	**Atributo**
3F00	posición vertical *sprite* #0
3F01	posición vertical *sprite* #1
3F02	posición vertical *sprite* #2
3F03	posición vertical *sprite* #3
3F04	posición vertical *sprite* #4
3F05	posición vertical *sprite* #5
3F06	posición vertical *sprite* #6
3F07	0xD0 (terminador)
.	
.	
.	
3F3F	posición vertical *sprite* #63
3F40	64 bytes sin usar.
.	
.	
.	
3F7F	
3F80	posición horizontal *sprite* #0
3F81	código del carácter *sprite* #0
3F82	posición horizontal *sprite* #1
3F83	código del carácter *sprite* #1
3F84	posición horizontal *sprite* #2
3F85	código del carácter *sprite* #2
.	
.	
.	
3FFE	posición horizontal *sprite* #63
3FFF	código del carácter #63

Fig. 21. Organización de Tabla de Atributos de *Sprites*

Esta tabla se iría leyendo de las direcciones inferiores a la superiores, de forma que un *sprite* situado en una posición superior de memoria podría tapar parcial o totalmente a otros *sprites* situados en posiciones inferiores de la tabla.

Si al analizar esta tabla se encontrara con el valor 0xD0 como atributo, se trataría como un separador de forma que se ignoraran todos los *sprites* que quedaran por recorrer en la tabla.

En la siguiente figura se muestra un esquema de la organización de la pantalla.

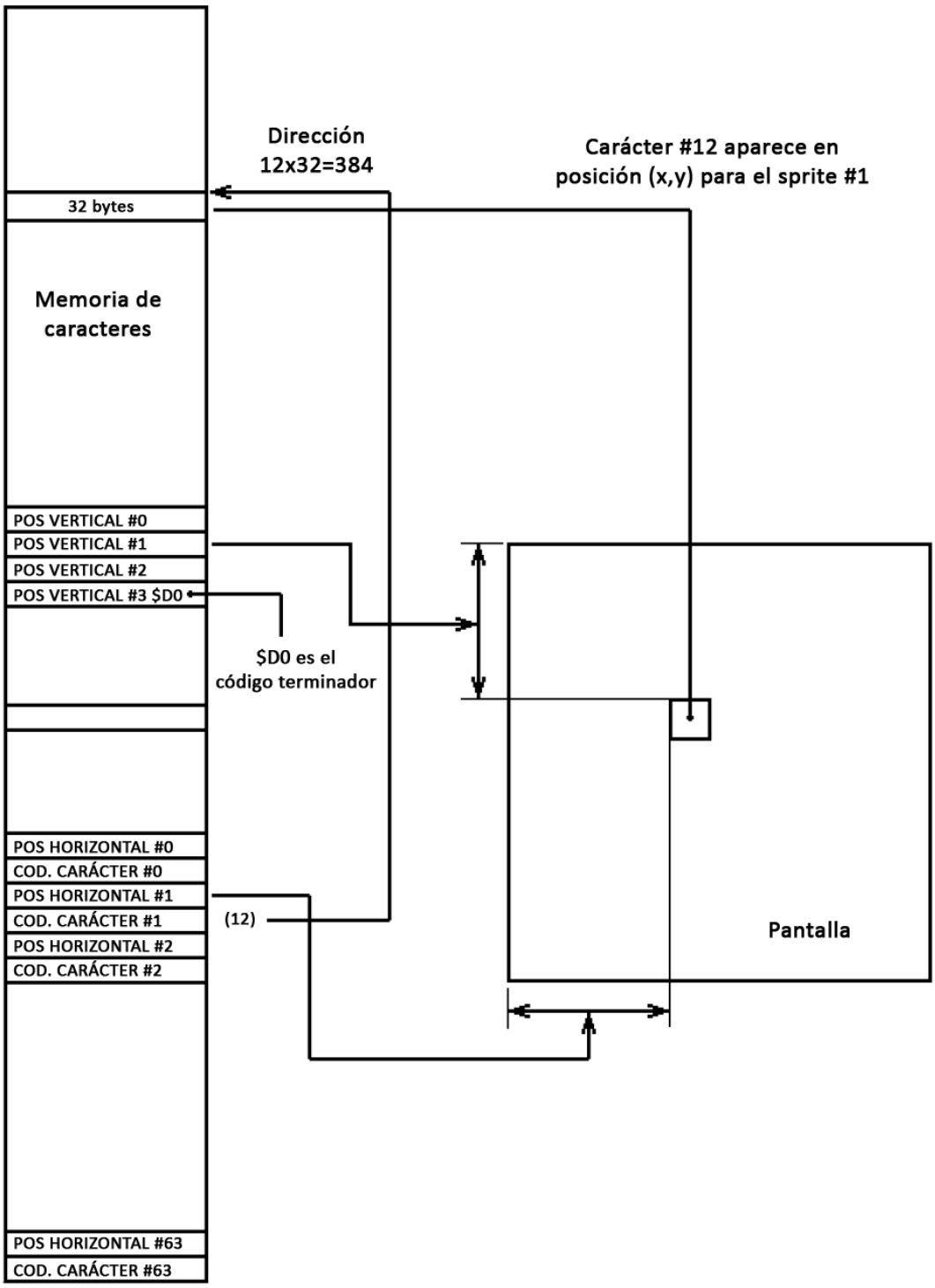


Fig. 22. Tabla de Atributos de *Sprites*

7. Prioridades entre capas.

Al existir solo dos capas gráficas las prioridades entre ellas son muy sencillas. Por un lado la prioridad entre *sprites* viene definida por su posición en la Tabla de Atributos de *Sprites*, mientras que cada *tile* tiene un bit en su estado que define si dicho *tile* se encuentra por encima o debajo de la capa de *sprites*.

Lo más normal es que los *tiles* se encuentren por debajo de los *sprites*, pues es lo más lógico para representar el escenario y sobre él los actores, pero en ocasiones se puede dar el caso de querer representar un objeto del escenario en primer plano, superponiéndose a los actores.

En la siguiente figura se muestran unas capturas de pantalla del emulador implementado mostrando tan solo la capa de *tiles*, la capa de *sprites* y ambas a la vez. Cabe destacar las palmeras, *tiles* con más prioridad, que se superponen a los *sprites* (el personaje de *Sonic* y el marcador oculto en la palmera).

Capa de *Tiles*

Capa de *Sprites*

Capa de *Tiles* + *Sprites*

Fig. 23. Prioridades entre capas

8. Acceso al VDP.

El Z80 tiene acceso al VDP a través de dos puertos de entrada/salida: 0xBE y 0xBF, que se comunican respectivamente con los registros de datos y comando/estado (comando en caso de una escritura y estado en caso de una lectura).

En el caso de una lectura del registro de comandos, éste devuelve el registro de estado del VDP y además borra las peticiones de interrupción del VDP.

En el caso de una escritura en el registro de comandos, ésta debe hacerse mediante dos accesos al puerto asociado. Los dos bits superiores del segundo byte escrito determinan una de cuatro posibles operaciones:

```
b7 b6 Operación
 0  0 Lectura de la VRAM
 0  1 Escritura en la VRAM
 1  0 Escritura en registro del VDP
 1  1 Escritura en la CRAM
```

Fig. 24. Comandos del VDP

Estas operaciones tienen el siguiente formato:

Lectura de la VRAM

Primer byte	A7	A6	A5	A4	A3	A2	A1	A0
Segundo byte	0	0	A13	A12	A11	A10	A9	A8

Escritura en la VRAM

Primer byte	A7	A6	A5	A4	A3	A2	A1	A0
Segundo byte	0	1	A13	A12	A11	A10	A9	A8

Una vez se han enviado los dos bytes al puerto de comandos con la operación y la dirección de memoria sobre la que actuar (A13-A0 representa los 14 bits de esta dirección), se puede leer o escribir en el puerto de datos.

El VDP incorpora un mecanismo de autoincremento de las direcciones, de forma que, una vez establecida la dirección para leer o escribir, ésta se incrementará en una unidad con cada acceso, evitando así el tener que establecer su dirección para cada dato a transferir.

Escritura en registro del VDP

Primer byte	D7	D6	D5	D4	D3	D2	D1	D0
Segundo byte	1	0	0	0	R3	R2	R1	R0

En el primer byte se enviaría el dato a escribir en el registro y en el segundo el número del registro (indicado por los bits R3-R0) en el cual se escribirá.

Cabe destacar que al ir el dato incluido en el propio comando no hay que escribir nada en el registro de datos.

Escritura en la CRAM

Primer byte	0	0	0	A4	A3	A2	A1	A0
Segundo byte	1	1	0	0	0	0	0	0

Una vez enviado el par de bytes correspondientes a la dirección de la CRAM donde se escribirá (dada por los bits A4-A0) se enviará un nuevo dato esta vez al registro de datos con el color a ser escrito en la forma:

Color a CRAM	0	0	B1	B0	G1	G0	R1	R0

Donde B1-B0, G1-G0 y R1-R0 se corresponden respectivamente a la intensidad del color azul, verde y rojo que dará como resultado el color final.

Al igual que sucedía con la VRAM, la CRAM también es autoincrementable.

9. Registros del VDP.

El VDP cuenta con 12 registros necesarios para establecer las diferentes configuraciones y modos de funcionamiento.

A continuación se detalla cada uno de estos registros por separado.

Registro 0 y Registro 1

Estos dos registros establecen el modo de video e interrupciones del VDP.

Reg. 0	Bit	Valor	Detalles
	b7	VSI	Inhibición de desplazamiento vertical (últimas 8 columnas). 0: Se desplazan junto con el resto del fondo. 1: Se mantienen fijas (no se desplazan).
	b6	HSI	Inhibición de desplazamiento horizontal (primeras 2 filas). 0: Se desplazan junto con el resto del fondo. 1: Se mantienen fijas (no se desplazan).
	b5	LCB	Primera columna en blanco. 0: Modo normal. 1: Pinta toda la primera columna del color del borde.
	b4	IE1	Habilita interrupción *raster line*.
	b3	SS	Desplazamiento de *sprites.* 0: Posiciones normales de *sprites*. 1: Desplaza todos los *sprites* 8 píxeles a la izquierda.
	b2	1	Siempre debe estar a 1.
	b1	1	Siempre debe estar a 1.
	b0	ES	Sincronismo externo (siempre debe estar a 0).
Reg. 1	**Bit**	**Valor**	**Detalles**
	b7	1	Siempre debe estar a 1.
	b6	DIS	Pantalla encendida. 0: No muestra nada por pantalla. 1: Modo normal.
	b5	IE	Habilita interrupción *VBLANK*.
	b4	0	Siempre debe estar a 0.
	b3	0	Siempre debe estar a 0.
	b2	0	Siempre debe estar a 0.
	b1	SZ	Tamaño de *sprites*. 0: Todos los *sprites* son de 8x8 píxeles. 1: Dobla el ancho de todos los *sprites* a 8x16 píxeles.
	b0	ZM	Zoom en *sprites*. 0: Todos los *sprites* tienen el tamaño indicado en SZ. 1: Todos los *sprites* doblan las dimensiones indicadas en SZ.

Registro 2

Este registro establece la dirección base del Mapa de Pantalla como se puede ver en la siguiente tabla:

Valor de R2	Dirección base del Mapa de Pantalla
0xF1	0x0000
0xF3	0x0800
0xF5	0x1000
0xF7	0x1800
0xF9	0x2000
0xFB	0x2800
0xFD	0x3000

0xFF	0x3800

Registro 3 y Registro 4

Las funcionalidades de estos registros no están documentadas y se aconseja inicializarlos siempre a 0xFF.

Registro 5

Este registro determina la dirección base de la Tabla de Atributos de *Sprites* de entre 64 diferentes, teniendo en cuenta que el byte inferior de la dirección será 0x00 y el superior viene dado por el contenido del registro de la siguiente forma:

1	A13	A12	A11	A10	A9	A8	1

Registro 6

Este registro determina la dirección base de la VRAM a partir de la cual se almacenarán los caracteres. Solo son válidos dos valores: 0xFB (primeros 8 Kbytes) y 0xFF (segundos 8 Kbytes).

Registro 7

Este registro determina el color del borde. El índice del color almacenado en este registro (C3-C0) se refiere al segundo banco de colores (el usado por los *sprites*) de la CRAM:

1	1	1	1	C3	C2	C1	C0

Registro 8

Este registro establece el valor del desplazamiento horizontal del fondo de la escena (la capa de *tiles*), de forma que un valor de 0x00 no produce desplazamiento y valores superiores mueven dicha cantidad de píxeles hacia la izquierda.

Cabe destacar que la pantalla se comporta como si fuera un cilindro, de forma que todos los píxeles que “desaparezcan” por el borde izquierdo de la pantalla debido a un desplazamiento “aparecerán” desplazados tantos píxeles por el borde derecho.

Registro 9

Este registro establece el valor del desplazamiento vertical del fondo de la escena de forma análoga al registro anterior, desplazando hacia la parte superior los píxeles.

En este caso hay que tener en cuenta las 4 filas de caracteres no visibles, pues también sufren de este desplazamiento.

Registro 10

Este registro controla cuándo se producen las interrupciones de tipo *raster line* tal y como se verá en el siguiente apartado.

Registro de estado

Este registro muestra el estado del VDP, siendo significativos tan solo los tres bits superiores de la forma:

Bit	Descripción
b7	Se pone a 1 si se ha generado una interrupción en el VDP.
b6	Se pone a 1 si hay más de 8 *sprites* en una misma línea.
b5	Se pone a 1 si se superponen (colisionan) dos o más *sprites*.

Cabe remarcar que cada vez que el Z80 haga una lectura sobre el puerto de comandos este registro se establecerá a cero.

Valores por defecto

Al iniciarse el VDP estos registros tomarán los siguientes valores:

Registro	Valor
R0	0x36
R1	0xA0
R2	0xFF
R3	0xFF
R4	0xFF
R5	0xFF
R6	0xFB
R7	0x00
R8	0x00
R9	0x00
R10	0xFF

10. Interrupciones en el VDP.

El VDP puede generar dos tipos diferentes de interrupciones, conocidas como *raster line* y *VBLANK*, que pueden ser habilitadas o inhibidas mediante los registros 0 y 1.

La interrupción *VBLANK* se produce cada vez que se termina de pintar un fotograma, es decir, cada 192 líneas.

Por su parte la interrupción *raster line* se produce cada vez que un contador asociado al registro 10 se pone a cero. Este contador se carga con el valor del registro 10 y se decrementa cada vez que se pinta una línea, recargándose nuevamente al llegar a cero.

Las interrupciones del VDP serán muy importantes para la Master System, pues gracias a las *VBLANK* los programas pueden llevar una temporización de forma que pinten en la pantalla solo cuando ésta ha terminado de refrescarse (evitando así distorsiones en la imagen) y con la *raster line* se pueden hacer "trucos" gráficos como doblar el número de colores visibles simultáneamente al cambiar la paleta a la mitad del pintado de la pantalla.

11. Implementación.

El VDP se ha implementado como la clase mostrada en la figura 25.

Al igual que para el caso del Z80 se ha usado una estructura contextoVDP, mostrada en la figura 26, que permitirá almacenar el estado actual de forma compacta y así poder guardarlo y cargarlo de disco para el caso de los *savestates*.

El aspecto más destacable de la implementación realizada es la utilización de una caché de caracteres.

La decodificación de los índices de colores de los caracteres, al estar codificados mediante cuatro planos, es muy ventajosa para el hardware, pero muy costosa de emular, pues habría que decodificarlo en cada fotograma. Por este motivo se ha optado por crear una pequeña caché auxiliar con todos los caracteres de la VRAM decodificados, de modo que cada vez que se escriba un carácter en la memoria del VDP su posición sea marcada como "sucia" en la caché, para ser decodificado cuando vaya a ser pintado, pero aprovechando todos los otros caracteres decodificados hasta el momento para ahorrar un costoso tiempo de proceso.

Por otro lado, las direcciones base de zonas de memoria como la Tabla de Atributos de *Sprites* o el Mapa de Pantalla se han codificado usando punteros de forma que si en tiempo de ejecución se modificaran los registros que establecen estas direcciones, fuera instantáneo su cambio de posición.

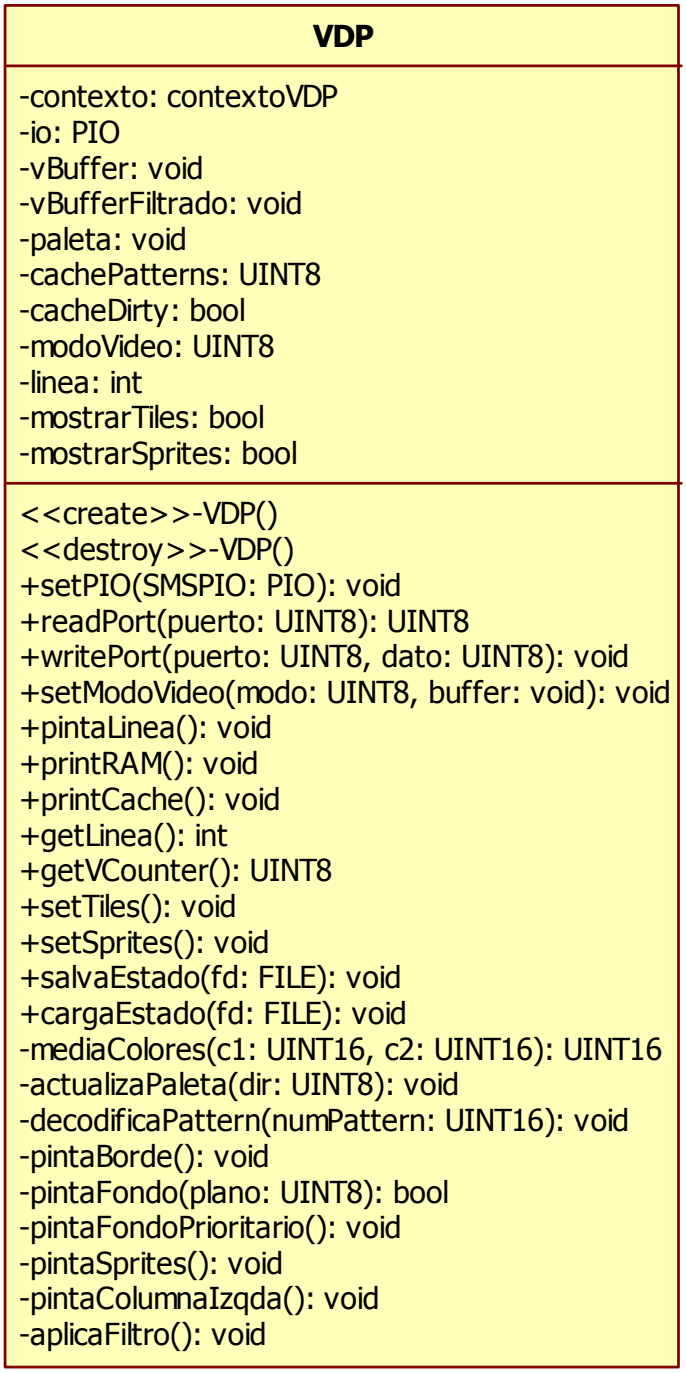


Fig. 25. Clase VDP

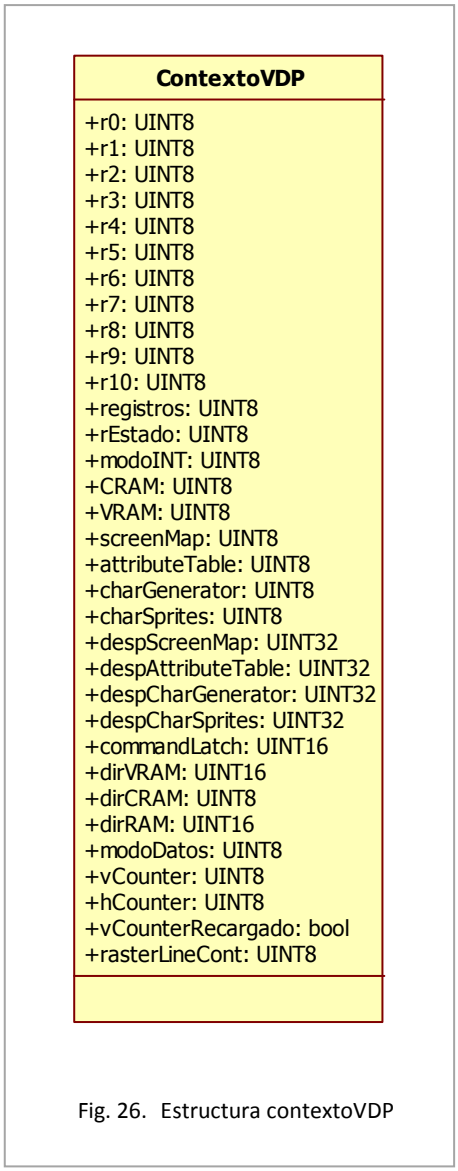


Fig. 26. Estructura contextoVDP

En cuanto al tema del pintado de los caracteres en pantalla se ha realizado copiando los píxeles de éstos en las posiciones adecuadas del buffer asociado a la ventana gráfica del emulador.

Este proceso es sencillo y relativamente eficiente cuando se trata de copiar los caracteres tal cual, pues se puede realizar copiando bloques de 8 píxeles contiguos en memoria, usando instrucciones de copias de memoria (*memcpy* en C/C++), en las posiciones adecuadas, pero cuando se trata de caracteres con “efectos especiales” ya no es posible este método y hay que recurrir a calcular la posición unitaria de cada píxel y copiarlo uno a uno.

En la Master System estos “efectos especiales” se reducen básicamente a dos: reflejados y escalados.

Un reflejado consiste en representar un carácter como si estuviera reflejado en un espejo, usando un eje de simetría vertical u horizontal para crear un reflejado de dicho tipo.

Su implementación se realizaría copiando las filas de 8 píxeles en sentido inverso si se trata de un reflejado horizontal o bien copiando las filas de 8 píxeles tal cual, pero invirtiendo la posición donde se copian, empezando de abajo arriba en lugar de arriba a abajo, si se trata de un reflejado vertical.

En la siguiente figura se puede apreciar este efecto sobre un carácter que representa la letra “P”.

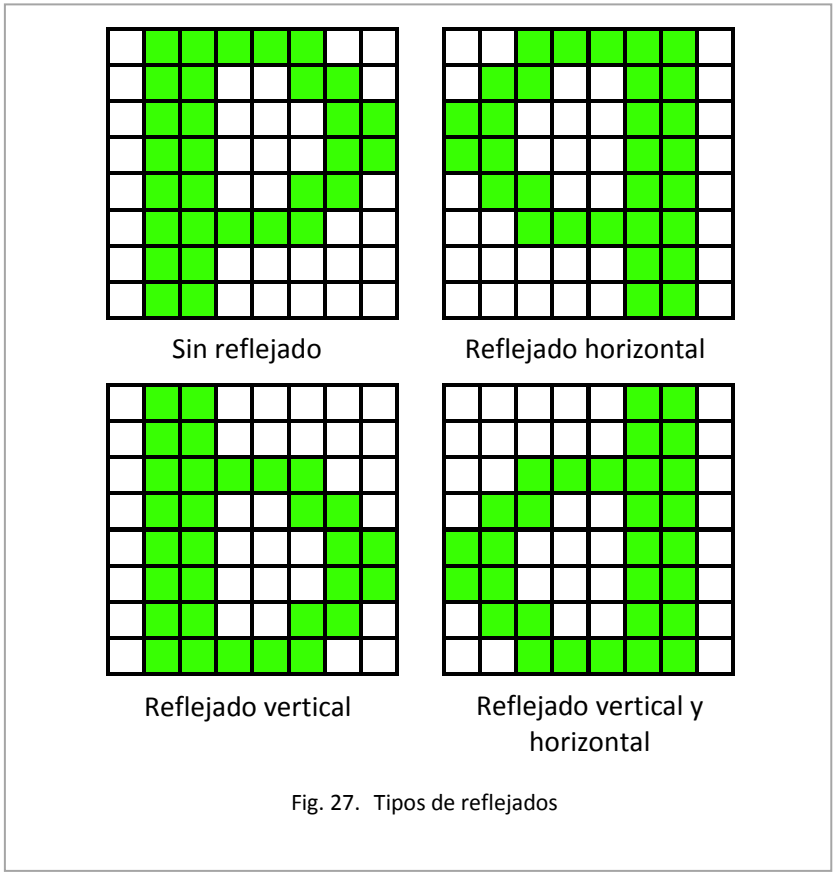


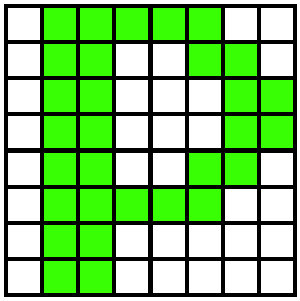
Fig. 27. Tipos de reflejados

Por su parte los caracteres también pueden sufrir dos tipos de escalados, esto es, dos formas de aumentar la imagen.

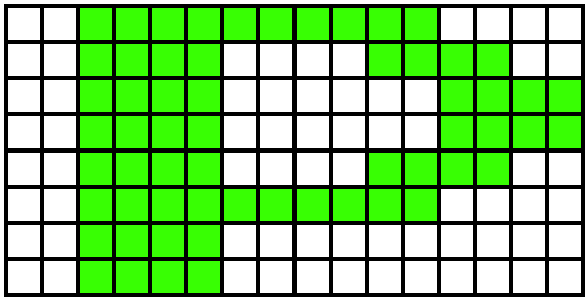
En el primero de ellos se haría un escalado parcial, se doblaría el ancho del carácter, dejando el alto tal cual, pasando a medir 16x8 píxeles en lugar de 8x8. En el segundo caso se realizaría un escalado total resultando una nueva imagen de 16x16 píxeles.

La forma de implementar estos escalados sería duplicando los píxeles de cada fila, escribiendo en pantalla dos píxeles iguales por cada uno leído, en ambos escalados, y duplicando además cada fila en caso de tratarse de un escalado total.

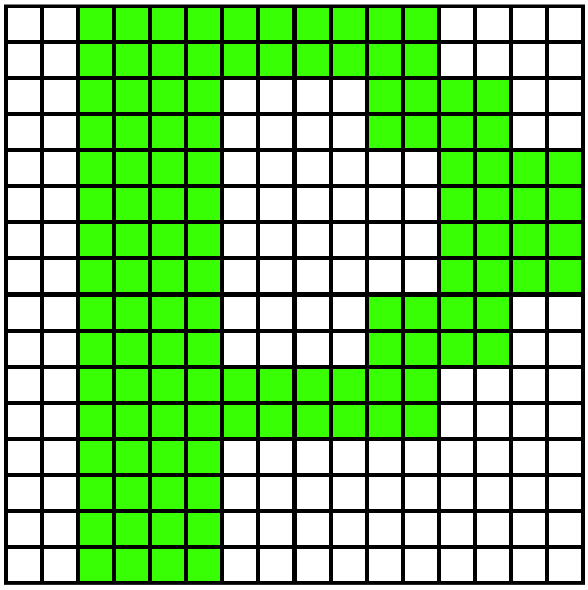
En la siguiente figura se muestra este efecto con el mismo ejemplo del caso anterior.



Sin escalado



Escalado parcial



Escalado total

Fig. 28. Tipos de escalados

Puesto que en la Master System contamos con dos procesadores (Z80 y VDP) y en el emulador creado para PC sólo tendremos uno (en el caso más básico de sistemas monoprocesador), hay que recurrir al reparto de recursos para emular estos dos chips.

Se podría haber hecho uso de hilos o procesos ligeros para emular cada procesador en uno de ellos, pero se ha optado por otra solución: entrelazar la ejecución.

El Z80 funciona a 3,579545 MHz, esto es, ejecuta 3.579.545 ciclos de reloj por segundo. El VDP muestra 60 fotogramas por segundo y en cada uno de ellos tiene que pintar 262 líneas de la pantalla (192 de las cuales son visibles y el resto invisibles), luego es sencillo deducir que se ejecutarán 228 ciclos de reloj entre cada línea pintada.

De este modo se llamará a ejecutar al Z80 durante 228 ciclos y acto seguido el VDP será invocado para pintar la línea que corresponda (se guarda un contador) tras lo que volverá a entrar en ejecución el Z80 y así cíclicamente.

A la hora de pintar cada línea primero se pintarán los *tiles* que correspondan dependiendo de su posición, comprobando si hay algún *tile* con el bit de prioridad activo. Tras pintarse los *tiles* se hará lo mismo con los *sprites* que se sitúen sobre dicha línea. Por último, si hubiera *tiles* prioritarios, se procedería a pintar estos de forma que quedaran siempre por encima de la capa de *sprites*.

Finalmente, se ha incorporado en esta clase una serie de filtros gráficos para reescalar la imagen en el PC al doble de su resolución nativa, de forma que si bien desvirtúa la imagen original, ofrece un mejor aspecto al presentarse en PCs con una resolución varias veces superior a la original. Concretamente se han añadido los filtros "2x *raw*", "2x con *scanlines*" y "2x bilinear".

El filtro "2x *raw*" reescala la imagen al doble de su tamaño simplemente doblando el tamaño de los píxeles a representar, de modo que cada píxel de la imagen original (1x1) se convierte en cuatro píxeles iguales (2x2) en la imagen final.

Este escalado da como resultado una imagen algo pixelada, pero su implementación consume menos recursos que los otros filtros.

El filtro "2x con *scanlines*" reescala la imagen generando un nuevo píxel entre cada dos píxeles originales. Estos nuevos píxeles son del color resultante de hacer la media entre los componentes de color de los dos píxeles a sus lados, de forma que los cambios de color resultan suavizados.

Se generarán tantas líneas como tenga la imagen original, pero entre cada una de ellas se intercalará una línea de color negro (sumando así el doble de líneas buscadas) que da un aspecto similar a las *scanlines* de los monitores de rayos catódicos.

Este método es bastante más costoso en tiempo de cálculo que el anterior debido a tener que hacer las medias de los colores (separando además sus componentes de color), pero tiene un cierto ahorro en las líneas negras en las que no hay que calcular nada.

Por último, el filtro "2x bilinear" es similar al anterior, pero en este caso se generan también píxeles nuevos entre cada una de las líneas de la imagen original a partir de las medias de los píxeles inferior y superior.

No cabe duda de que este método es el más costoso, pero la imagen obtenida es la que generalmente mejor aspecto presenta por su suavizado.

A continuación se muestran unas imágenes como ejemplo de estos filtrados:

Fig. 29. Video sin filtrar

Fig. 30. Video con filtrado *raw*

Fig. 31. Video con filtrado *scanlines*

Fig. 32. Video con filtrado bilinear

Capítulo 6. El generador de sonido programable (PSG).

1. Descripción general.

La Master System incorpora un chip dedicado exclusivamente a las funciones sonoras conocido como el generador de sonido programable o PSG (del término inglés *Programmable Sound Generator*).

Este chip está basado en el SN76489 fabricado por Texas Instruments e incorpora cuatro canales de sonido consistentes en tres generadores de tonos y un generador de ruido. Cada uno de estos canales tiene asociado su propio atenuador, permitiendo variar su volumen de forma independiente.

Como se puede ver en la figura siguiente, el PSG está conectado al reloj central de la Master System, dividiendo su frecuencia entre 16 y usando la frecuencia resultante (de unos 223 KHz) como base para configurar la frecuencia de los distintos canales.

Cada canal generará ondas cuadradas con unas frecuencias entre 109 Hz y 112 KHz con 16 escalas distintas de volumen.

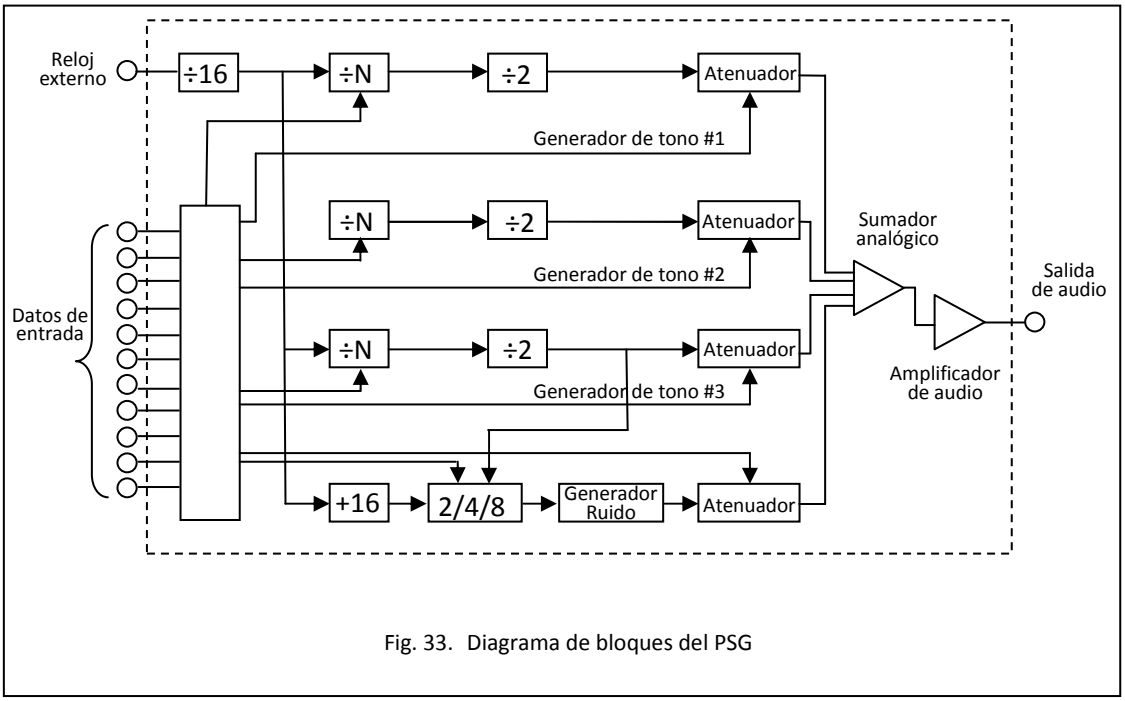


Fig. 33. Diagrama de bloques del PSG

2. Los canales del PSG.

De los cuatro canales incorporados en el PSG habría que hacer una distinción entre los canales de tono y el canal de ruido.

Los canales de tono

Cada uno de los tres canales de tono que incorpora el PSG será usado para producir sonidos armónicos: músicas, voces digitalizadas, efectos sonoros, etc.

Cada uno de estos canales producirá una onda de salida cuadrada que podrá ser modulada en frecuencia de acuerdo con el valor precargado en un contador con decremento. Este contador se irá decrementando en una unidad con cada pulso de reloj (en el caso del PSG de la Master System un dieciseisavo del reloj de ésta) y al llegar a cero se invertirá la salida del canal (de 0 a 1 y de 1 a 0) y se volverá a cargar el contador con el valor anterior u otro nuevo si así se especificara.

Estos registros contadores tienen una longitud de 10 bits, luego la frecuencia de entrada a cada canal puede ser dividida entre 1.024 distintos valores para producir otras tantas frecuencias de salida como se ve en la siguiente fórmula:

$$Frecuencia\ (Hz) = \frac{3.579.545\ Hz}{32\ \times valor\ registro}$$

Fig. 34. Cálculo de la frecuencia del PSG

Por ejemplo, un valor del registro de 0xFE resultaría en una frecuencia de unos 440 Hz. Por otro lado, un valor del registro de 0x0 resultaría en una salida constante de una onda cuadrada siempre a nivel alto.

Una vez generados los pulsos de salida, son filtrados a través del atenuador correspondiente a cada canal que multiplica su valor por uno de 16 posibles valores (estos atenuadores son registros de 4 bits) para variar la amplitud de la onda (su volumen) y su salida se mandará al mezclador.

El canal de ruido

Además de los canales de tono, el PSG incorpora un canal de ruido que producirá ruido blanco o periódico, usado para generar el sonido de percusiones, explosiones, interferencias, etc.

Sobre este canal, al igual que en los de tono, también actúa un divisor de frecuencia, en este caso seleccionable entre dos, cuatro, ocho o bien el mismo valor del canal de tono número tres, de acuerdo a los dos bits inferiores del registro asociado de la forma:

Bits inferiores del registro	Valor con el que se recargará el contador
00	0x10
01	0x20
10	0x40
11	Valor del registro de tono #3

La principal diferencia respecto a los canales de tono es que esta frecuencia de salida en lugar de pasar directamente por el atenuador correspondiente se usa como entrada para un registro de desplazamiento con retroalimentación lineal (LFSR) de 16 bits que será el encargado de generar el ruido.

Finalmente el ruido generado pasará a través del atenuador para regular su volumen como en el caso de los canales de tono y se pasará al mezclador.

Mezcla de canales

Una vez producida la salida de los cuatro canales y modificado su volumen con los atenuadores pasarán a través del mezclador.

Este mezclador es un sumador analógico con la única función de sumar las cuatro ondas resultantes para producir la onda final. Ésta se pasará a su vez a través de un amplificador de audio para producir la salida final que se enviará a través de las salidas que incorpora la Master System para ser reproducida en los altavoces del televisor o en unos audífonos.

3. Registros y programación del PSG.

El PSG cuenta con ocho registros: cuatro registros de 4 bits para el volumen, tres registros de 10 bits para la frecuencia de los canales de tono y un registro de 4 bits (aunque realmente solo se usan 3 de ellos) para el canal de ruido.

Inicialmente todos los registros de volumen se inicializarán a un valor de 0xF (silencio total) y los de tono a 0x0 (inactivos).

El procesador Z80 podrá comunicarse con el PSG a través de sus puertos de entrada/salida, concretamente a través del puerto 0x7F (o, menos frecuentemente, el puerto 0x7E). Esta comunicación es solo unidireccional, es decir, el Z80 podrá enviar órdenes al PSG, pero no hay manera de leer el estado o datos de éste.

Dependiendo de si el dato escrito en el puerto del PSG tiene su bit superior a 0 o 1 se tratará como un byte de datos o bien un byte de *latch*/datos.

En el caso de un byte de *latch*/datos éste tendrá el siguiente formato:

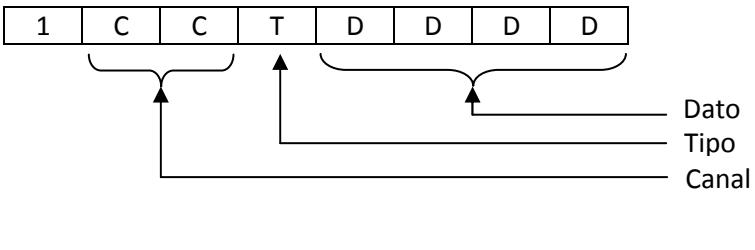


Fig. 35. Byte de *latch*/datos

Los bits 5 y 6 determinan el canal que será seleccionado de acuerdo a la siguiente tabla:

Canal	Registros de tono/ruido	Registros de volumen
00	Tono #1	Volumen #1
01	Tono #2	Volumen #2
10	Tono #3	Volumen #3
11	Ruido	Volumen #4

El bit número 4 determina el tipo de registro que se seleccionará de la tabla anterior, bien de tono/ruido si vale 0 o bien de volumen si vale 1.

Finalmente los bits 3 a 0 contienen el valor de los 4 bits de menor peso que serán copiados al registro seleccionado.

En el caso de un byte de datos éste tendrá el siguiente formato:

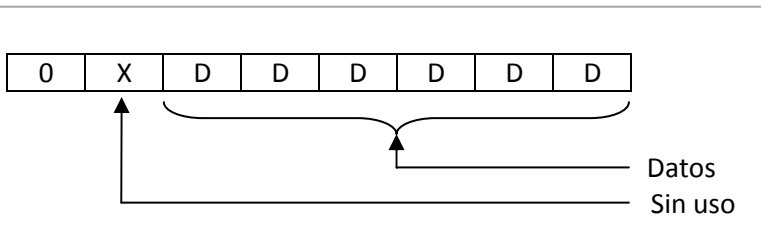


Fig. 36. Byte de datos

Donde los 6 bits de menor peso se corresponden con el dato que se copiará en los 6 bits de mayor peso del registro seleccionado en el byte de *latch*/datos.

Por tanto, para escribir en un registro de tono hará falta escribir dos bytes en el puerto 0x3F (para llenar su registro de 10 bits), mientras que si lo que se desea es cambiar el valor de un registro de volumen, con escribir un byte de *latch*/datos será suficiente pues solo cuenta con 4 bits.

En la siguiente figura se puede observar la atenuación obtenida dependiendo de los cuatro bits escritos.

A3	A2	A1	A0	Atenuación
0	0	0	0	0 dB (máximo volumen)
0	0	0	1	2 dB
0	0	1	0	4 dB
0	0	1	1	6 dB
0	1	0	0	8 dB
0	1	0	1	10 dB
0	1	1	0	12 dB
0	1	1	1	14 dB
1	0	0	0	16 dB
1	0	0	1	18 dB
1	0	1	0	20 dB
1	0	1	1	22 dB
1	1	0	0	24 dB
1	1	0	1	26 dB
1	1	1	0	28 dB
1	1	1	1	Canal silenciado

Fig. 37. Valor de los atenuadores

4. Implementación.

El PSG se ha implementado como la clase mostrada en la figura 38.

Al igual que en clases anteriores se ha usado una estructura contextoPSG para guardar el contexto y así poder salvar y cargar los *savestates* con mayor facilidad.

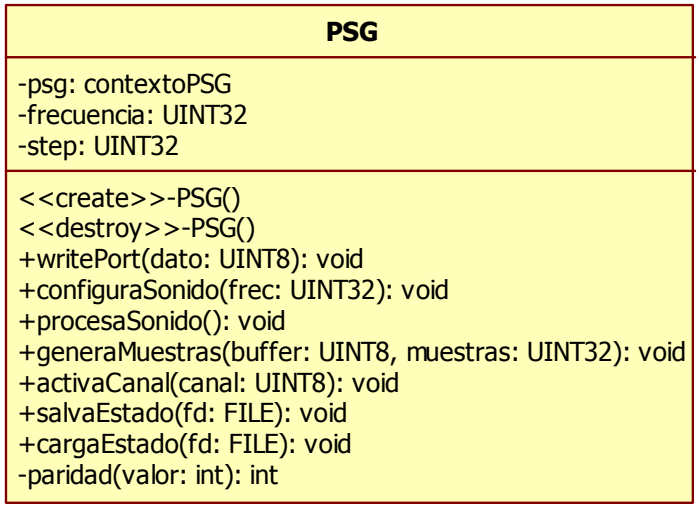


Fig. 38. Clase PSG

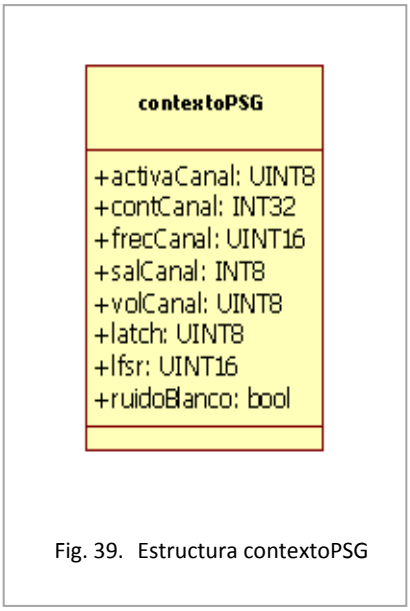


Fig. 39. Estructura contextoPSG

La función básica de generar muestras del sonido en el PSG la realiza el método *generaMuestras*.

Para reproducir el sonido en el PC se han empleado las bibliotecas multimedia SDL. Éstas emplean un buffer de datos que se envía periódicamente a la tarjeta de sonido para su reproducción, llamando a una función de tipo *callback* (funciones que se invocan automáticamente cada vez que sucede un evento) cada vez que se vacía el buffer para volver a llenarlo.

Esta función *callback* se ejecutará en un hilo o proceso ligero independiente para evitar problemas de sincronización con la ejecución y será la encargada de llamar al método *generaMuestras* para rellenar el buffer de sonido con las muestras generadas por éste.

Dependiendo de la frecuencia de muestreo (el número de muestras por segundo) seleccionada para el emulador (en el caso implementado se puede elegir entre 11.025 Hz, 22.050 Hz o bien 44.100 Hz) se obtendrá una calidad de sonido mejor o peor (cuanto mayor sea la frecuencia más muestras de la onda se tomarán y, por tanto, sonará con mayor calidad).

Conociendo la frecuencia de muestreo se conocerá a su vez el intervalo de tiempo entre cada muestra, con lo que al ser conocida la velocidad del reloj del PSG se tendrá el valor del "paso" o número de decremento del contador de cada canal que habrá que aplicar entre cada muestra tomada para seguir una correcta sincronización.

En la implementación realizada, debido a la cantidad de operaciones que se realizarán por segundo, tanto los contadores como los registros se han representado como datos enteros sin signo, en lugar de números en coma flotante, para incrementar la velocidad. Al tener que hacer cálculos con divisiones enteras se pierde parte de la pre-

cisión, sobre todo al dividir números pequeños, es por este motivo por el que se ha multiplicado tanto el valor del paso como el valor de los contadores por 4.096 para conseguir así una precisión extra a costa de un mínimo incremento de coste computacional. Se ha elegido este valor por ser lo mayor posible y aun así no provoque desbordamientos de los registros al multiplicar los valores que en ellos hubiera.

En la figura 40 se puede ver el código en detalle de la función *generaMuestras*.

Conociendo el tamaño del paso, al dividirlo por la frecuencia del canal se conocerá el número de cambios que ha realizado la onda de salida de nivel alto a bajo o viceversa. Como en la implementación realizada se utilizan muestras con signo (el nivel alto correspondería a 1 y el bajo a -1) si el número de cambios de nivel realizados es par la salida seguiría siendo la misma y se tomaría tal cual como la muestra buscada, sin embargo, si el número de cambios fuera impar la salida sería la inversa de la actual y sería la que se tomara como muestra.

Una vez calculada la muestra, se amplificará dependiendo del valor del volumen del canal en cuestión, por lo que se hará el producto de estos dos valores (es el equivalente a una amplificación).

La salida del canal de ruido se realizará de modo análogo, pero teniendo en cuenta que en este caso se usará el resultado de una función de paridad (para el caso de ruido blanco) o del LFSR (para el caso del ruido periódico) como salida. También hay que tener en cuenta que estas funciones usan muestras sin signo (1 para el nivel alto y 0 para el bajo) luego si el número de cambios es impar se realizará la función XOR en lugar de la negación para invertir el resultado.

Finalmente, las muestras generadas en cada canal se pasarán al mezclador, cuya salida en la práctica no es más que la suma aritmética de éstas.

```
void PSG::generaMuestras (UINT8 * buffer, UINT32 muestras)
{
   INT16 muestra = 0;
   INT16 muestraTemp;
   UINT32 canal, cambios;
   INT16 * p = (INT16 *) buffer;
   while (muestras > 0)
   {
      /* Canales de tonos */
      for (canal = 0; canal < 3; canal++)
      {
         if (!psg.activaCanal[canal]) continue;
         if (psg.frecCanal[canal] != 0)
         {
            psg.contCanal[canal] += step;
            if (psg.contCanal[canal] >= (psg.frecCanal[canal]<<12))
            {
               cambios = psg.contCanal[canal] /
                             (psg.frecCanal[canal]<<12);
               psg.contCanal[canal] -= (psg.frecCanal[canal]<<12) *
                                       cambios;
               if (cambios & 0x01)
                  psg.salCanal[canal] = -psg.salCanal[canal];
            }
         }
         muestraTemp = psg.salCanal[canal];
         muestraTemp *= (INT16) ( (15 - psg.volCanal[canal]) << 8);
         muestra += muestraTemp;
      }

      /* Canal de ruido */
      if (psg.activaCanal[3])
      {
         if (psg.frecCanal[3] != 0) psg.contCanal[3] += step;
         while ((psg.contCanal[3] >= psg.frecCanal[3]<<12) &&
                                     (psg.frecCanal[3] != 0))
         {
            psg.contCanal[3] -= psg.frecCanal[3]<<12;
            psg.salCanal[3] = ~psg.salCanal[3];
            if (psg.salCanal[3])
            {
               psg.lfsr = (psg.lfsr>>1)
                           | ((psg.ruidoBlanco ? paridad(psg.lfsr &
                             0x9): psg.lfsr & 1)<<15);
               muestraTemp = (psg.lfsr & 1);
               muestraTemp *= (INT16) ( (15 - psg.volCanal[3])
                                                        << 8);
               muestra += muestraTemp;
            }
         }
      }
      *p = muestra;
      ++p;
      *p = muestra;
      ++p;
      muestra = 0;
      muestras--;
   }
}
```

Fig. 40. Función *generaMuestras*

Capítulo 7. Los dispositivos controladores.

1. Descripción general.

Al tratarse la Master System de una videoconsola, todos sus dispositivos controladores están diseñados específicamente para responder al control de juegos, de ahí que sean sumamente sencillos para que cualquier persona se pueda adaptar a ellos de inmediato sin ningún problema.

Todos los modelos de Master System incorporan dos puertos para mandos, de forma que sea posible jugar dos personas en la misma videoconsola.

El mando de control estándar incorporado de serie con la Master System, como se puede apreciar en la siguiente figura, presenta un diseño rectangular e incorpora dos botones de acción y un *pad* direccional de ocho direcciones (aunque internamente tiene cuatro contactos).

Fig. 41. Mando de control estándar de la Master System

A medida que se fue asentando en el mercado la Master System, tanto SEGA, su fabricante, como terceras compañías empezaron a sacar al mercado nuevos dispositivos como pistolas, *joysticks*, mandos más ergonómicos y otros dispositivos dispares, algunos de los cuales se pueden ver en la figura 43.

A pesar de esta disparidad de dispositivos, todos ellos tenían un interfaz físico común tal y como se ve en la siguiente figura.

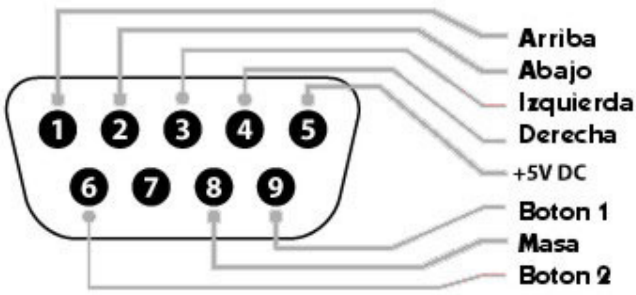


Fig. 42. Conexiones de los mandos de control

Fig. 43. Otros dispositivos de control

2. Puertos de entrada/salida.

Los dispositivos de control se comunican con el procesador Z80 a través de dos puertos de entrada/salida, concretamente los puertos 0xDC y 0xDD.

Estos puertos funcionan con lógica inversa, su estado en "reposo" sería con todos sus bits a un valor de 1 lógico, mientras que el contacto de cada uno de los microinterruptores incorporados en el mando (uno por botón o dirección del *pad*) provocará que un bit determinado se establezca con un 0 lógico.

La pulsación de cada uno de estos contactos no es excluyente, es decir, sería posible leer cualquier pulsación simultánea de los contactos siempre que ésta sea físicamente posible (debido a la propia construcción del mando estándar es imposible pulsar dos direcciones opuestas).

En la siguiente figura se muestra a qué dirección corresponde cada bit de estos puertos:

Puerto 0xDC	
Bit	**Acción**
7	Jugador 2 – Abajo
6	Jugador 2 – Arriba
5	Jugador 1 – Botón 2
4	Jugador 1 – Botón 1
3	Jugador 1 – Derecha
2	Jugador 1 – Izquierda
1	Jugador 1 – Abajo
0	Jugador 1 – Arriba

Puerto 0xDD	
Bit	**Acción**
7	Sin uso
6	Sin uso
5	Sin uso
4	Reset
3	Jugador 2 – Botón 2
2	Jugador 2 – Botón 1
1	Jugador 2 – Derecha
0	Jugador 2 – Izquierda

Fig. 44. Puertos de entrada/salida de los dispositivos controladores

Por ejemplo, si el Z80 leyera un valor de 0x25 en el puerto 0xDC lo interpretaría como que hay un movimiento del jugador 1 hacia la diagonal arriba-izquierda y al mismo tiempo se está pulsando el botón de acción 2.

3. Implementación.

Los dispositivos controladores se han implementado como la clase Joysticks que se puede ver en la siguiente figura.

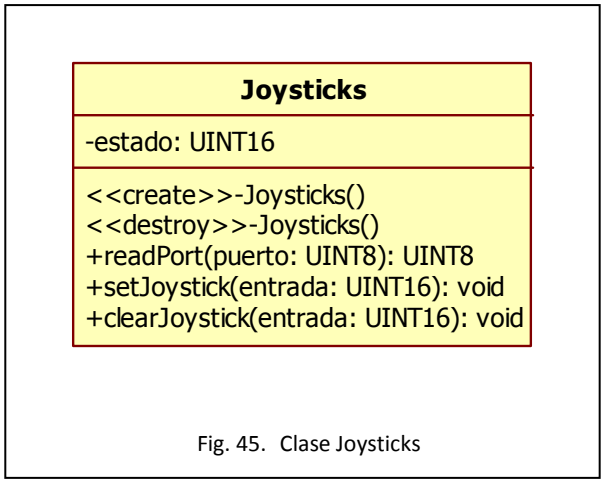


Fig. 45. Clase Joysticks

La implementación de esta clase es ciertamente trivial. Desde la clase principal (Master System) se detectarán por medio de las bibliotecas SDL los eventos generados al pulsar alguna tecla del teclado o bien de un *joystick* conectado al ordenador.

Cuando se detecte uno de estos eventos se invocará el método *setJoystick* de modo que se escriba en el objeto correspondiente la acción mapeada al evento generado (como marcar como activo el bit de Jugador 1 – Arriba si se pulsar la tecla de la flecha hacia arriba del teclado).

Para una mayor sencillez, el estado que se devolverá al leer de los dos puertos de entrada/salida se almacena en un único miembro del objeto (de nombre estado) de 16 bits. Al leer el Z80 de uno u otro puerto se enviarán los 8 bits correspondientes mediante un desplazamiento.

Análogamente a los valores de los puertos de entrada/salida, los bits del miembro estado tienen los siguientes significados:

Bit	Acción
15	Sin uso
14	Sin uso
13	Sin uso
12	Reset
11	Jugador 2 – Botón 2
10	Jugador 2 – Botón 1
9	Jugador 2 – Derecha
8	Jugador 2 – Izquierda

Bit	Acción
7	Jugador 2 – Abajo
6	Jugador 2 – Arriba
5	Jugador 1 – Botón 2
4	Jugador 1 – Botón 1
3	Jugador 1 – Derecha
2	Jugador 1 – Izquierda
1	Jugador 1 – Abajo
0	Jugador 1 – Arriba

Fig. 46. Estado interno de la clase Joysticks

Capítulo 8. El selector de programas.

1. Introducción y parámetros.

Si bien el emulador ha sido programado de forma que se lance desde la línea de comandos, para facilitar su uso se ha implementado una interfaz gráfica de usuario.

Desde esta interfaz se puede seleccionar de forma visual, a golpe de ratón, cualquiera de las opciones que de otra forma habría que pasar desde la línea de comandos.

En la siguiente figura se puede ver una captura de pantalla de la interfaz enumerando sus distintos elementos.

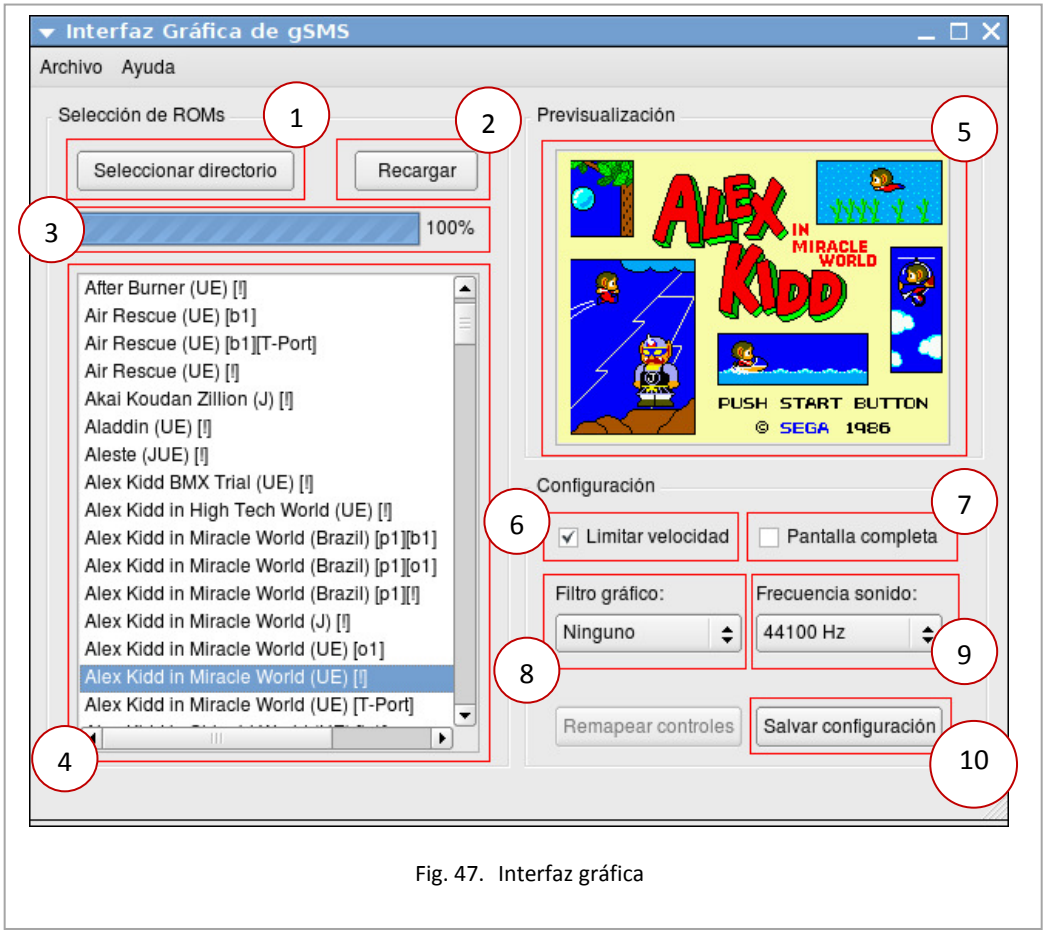


Fig. 47. Interfaz gráfica

A continuación se proporciona una breve explicación de cada apartado de la interfaz.

1. Desde este botón se puede elegir el dispositivo y directorio en el que se buscarán programas compatibles con la Master System (conocidos como ROMs por ser precisamente volcados de las memorias ROM contenidas en los cartuchos

originales de la consola). Para su correcto reconocimiento deberán tener extensión .sms
2. Este botón sirve para recargar el directorio activo donde se ubican los programas para el emulador. Al pulsarlo se volverá a analizar dicho directorio.
3. Esta barra de progreso muestra el estado actual del análisis del directorio iniciado al pulsar el botón Recargar.
4. En esta lista desplegable se mostrarán todos los programas encontrados en el directorio buscado. Se ha incluido una pequeña base de datos con el CRC (control de redundancia cíclica) de todos los programas comerciales conocidos, de modo que al detectar uno de ellos se sustituirá el nombre del archivo por el nombre real del programa almacenado en esta base de datos.
5. En esta zona se mostrará una captura de pantalla (si la hubiera) del programa seleccionado a modo de muestra para una mejor identificación. Estas capturas deberán tener el mismo nombre que el programa seleccionado y estar codificadas como imágenes de mapa de bits de Windows (BMP), desde el propio emulador se guardan automáticamente de esta forma al pulsar la tecla F10.
6. Esta casilla de verificación permite elegir entre limitar la velocidad de ejecución a la real de la Master System (60 fotogramas por segundo) o por el contrario ejecutar a tanta velocidad como permita la máquina donde se ejecute el emulador. Desactivar esta casilla puede ser útil para propósitos de depuración.
7. Esta casilla de verificación permite ejecutar el emulador a pantalla completa. La resolución exacta mostrada dependerá de la tarjeta gráfica instalada en el ordenador y las resoluciones que soporte ésta.
8. Aquí se puede elegir el filtro gráfico a usar de entre los siguientes: Ninguno, 2x *raw*, 2x con *scanlines* o 2x bilinear. Para una descripción de los mismos véase el capítulo dedicado al VDP.
9. En este desplegable se puede elegir la frecuencia de muestreo de la música y efectos sonoros entre 44100 Hz, 22050 Hz o 11025 Hz. Como se vio en el capítulo dedicado al PSG, cuanto mayor sea ésta mayor calidad tendrá el sonido resultante.
10. Al pulsar este botón se guardarán las opciones elegidas para usarse como configuración por defecto en las siguientes ejecuciones de la interfaz gráfica.

2. Teclas de función.

Desde el propio emulador se han incluido varias opciones de control o depuración controlables mediante atajos de teclado y teclas de función.

Si bien estas funciones no forman parte de la propia interfaz de usuario, sino del ejecutable del emulador, se ha decidido documentarlas aquí puesto que un usuario estándar normalmente no notará la separación entre ambas partes.

En la siguiente figura se detallan estas teclas de función y una breve descripción de su utilidad.

Atajo	**Función**
ESCAPE	Finaliza la ejecución del emulador.
F1	Selecciona el *slot* 1 como el actual para los *savestates*.
F2	Selecciona el *slot* 2 como el actual para los *savestates*.
F3	Selecciona el *slot* 3 como el actual para los *savestates*.
F4	Selecciona el *slot* 4 como el actual para los *savestates*.
F5	Selecciona el *slot* 5 como el actual para los *savestates*.
F6	Activa/Desactiva canal de tono 1 del PSG.
F7	Activa/Desactiva canal de tono 2 del PSG.
F8	Activa/Desactiva canal de tono 3 del PSG.
F9	Activa/Desactiva canal de ruido del PSG.
F10	Realiza una captura de pantalla.
F11	Activa/Desactiva capa de *tiles* del VDP.
F12	Activa/Desactiva capa de *sprites* del VDP.
CTRL+S	Salva un *savestate* en el *slot* actual.
CTRL+L	Carga un *savestate* del *slot* actual.
Arriba	Equivale a la dirección Arriba del mando 1.
Abajo	Equivale a la dirección Abajo del mando 1.
Izquierda	Equivale a la dirección Izquierda del mando 1.
Derecha	Equivale a la dirección Derecha del mando 1.
Z	Equivale al botón 1 del mando 1.
X	Equivale al botón 2 del mando 1.
Espacio	Equivale al botón de pausa de la Master System.
Keypad 8	Equivale a la dirección Arriba del mando 2.
Keypad 5	Equivale a la dirección Abajo del mando 2.
Keypad 4	Equivale a la dirección Izquierda del mando 2.
Keypad 6	Equivale a la dirección Derecha del mando 2.
O	Equivale al botón 1 del mando 2.
P	Equivale al botón 2 del mando 2.

Fig. 48. Teclas de función y atajos de teclado

3. Implementación.

La interfaz gráfica se ha implementado utilizando la biblioteca gráfica multiplataforma QT4 de forma que sea fácilmente portable a cualquier plataforma soportada por ésta (Windows, Linux, MacOS X…).

Aunque podría haberse implementado la interfaz gráfica en el mismo ejecutable que el emulador, se ha decidido hacerlo como dos ejecutables separados. De este modo se consigue una total independencia, pudiendo ejecutar el emulador sin interfaz desde un terminal de comandos y además se aísla cada ejecutable de posibles errores causados por el otro.

Cuando se seleccione un programa para su ejecución desde la interfaz, ésta hará un *fork* (una copia idéntica) de su proceso y a continuación cambiará la imagen de memoria de este nuevo proceso por la del emulador, pasándole como parámetros los elegidos desde el interfaz.

El proceso recién creado será lanzado en primer plano mientras que la interfaz permanecerá en segundo plano para volver a ella cuando finalice la ejecución.

Capítulo 9. Desarrollo, verificación y pruebas.

Para verificar el correcto funcionamiento del emulador implementado se realizaron varios conjuntos de pruebas, cada uno destinado a probar una fase del desarrollo.

Los primeros elementos en ser desarrollados fueron la implementación del Z80 y una versión básica de los módulos de puertos de entrada/salida y memoria. Al no tener desarrollado ningún módulo que mostrara salidas del emulador se implementó el simulador Z80Sim (ver Apéndice C para más detalles) para ver el estado interno de cada registro y posición de memoria.

Se escribieron varios programas en código ensamblador del Z80, uno por cada grupo de instrucciones de éste, donde se usaran todas y cada una de las instrucciones pertenecientes a dicho grupo. Una vez ensamblados dichos programas (usando el ensamblador de código libre Pasmo) fueron ejecutados en modo paso a paso desde el Z80Sim para comprobar que, efectivamente, cada registro, flag o posición de memoria era correctamente modificado y el secuenciamiento de instrucciones seguía el orden esperado.

Con el Z80 ya implementado y verificado en primera instancia su funcionamiento, se implementó una sencilla consola de depuración (realmente un puerto de entrada/salida en el que cada dato escrito se muestra por la salida estándar). Esta consola fue usada para comprobar el resultado de una demo técnica de libre distribución llamada ZEXALL. Este programa, escrito en código máquina del Z80, realiza unas pruebas exhaustivas de cada instrucción del Z80 y luego compara los resultados obtenidos con los esperados para informar de posibles errores y las instrucciones afectadas por éstos. Gracias a ZEXALL se encontraron errores en varios grupos de instrucciones que pasaron inadvertidos con el uso del Z80Sim, así que fueron corrigiéndose hasta superar esta prueba correctamente.

```
SDSC Terminal de depuracion.
----------------------------
Z80 instruction exerciser

ld hl,(nnnn).................OK
ld sp,(nnnn).................OK
ld (nnnn),hl.................OK
<daa,cpl,scf,ccf>............   CRC:681be242 expected:9b4ba675
aluop a,(<ix,iy>+1)..........OK
aluop a,<ixh,ixl,iyh,iyl>....OK
aluop a,<b,c,d,e,h,l,(hl),a>.   CRC:b2fcb670 expected:5ddf949b
```

Fig. 49. Algunas líneas de salida de ZEXALL mostrando errores detectados

Tras asegurarse del correcto funcionamiento del Z80 se procedió a implementar el VDP para poder obtener resultados gráficos del emulador.

Con el VDP implementado se probaron varias demos técnicas de la Master System, desarrolladas por otros aficionados a la emulación, que exploraban todas las posibilidades gráficas de ésta (escalados, cambios de paletas, desplazamientos verticales y horizontales, pintado por capas, etc.). Nuevamente surgieron varios errores gráficos que fueron corrigiéndose poco a poco.

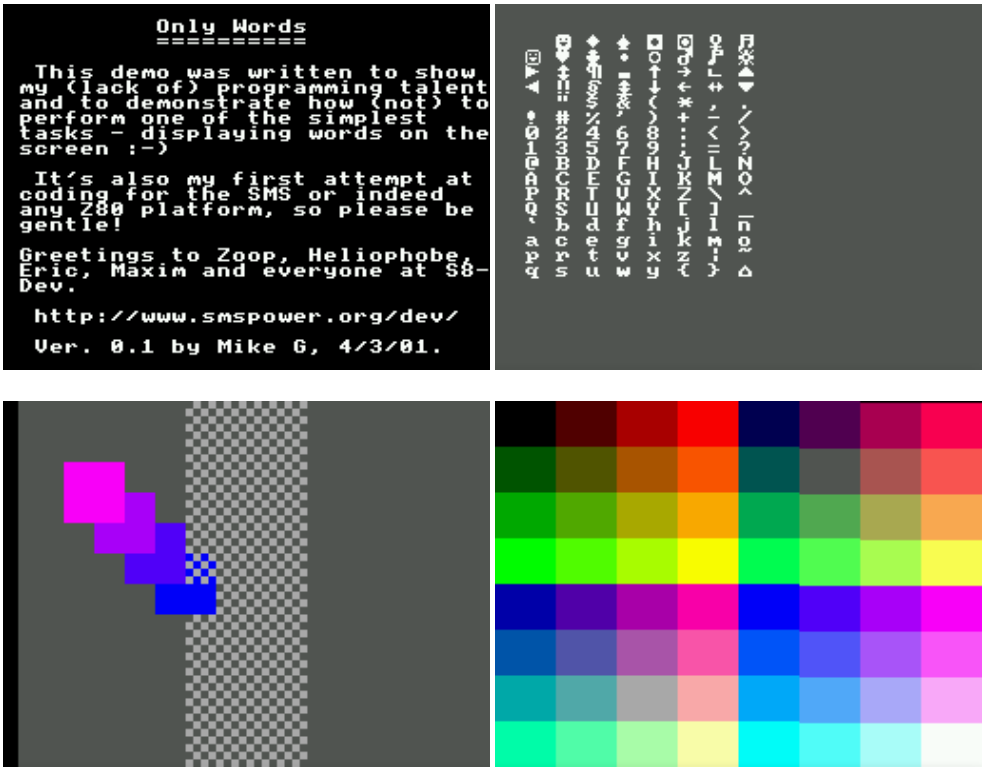


Fig. 50. Varias demos gráficas para verificar el VDP

Tras probarse el funcionamiento del VDP se implementaron los dispositivos controladores y mapeadores de memoria. A continuación se usó el VDP como apoyo para ver los resultados de otras demos técnicas que verificaban el correcto funcionamiento de estos dispositivos.

Fig. 51. Demos para verificar controladores y mapeadores de memoria

Una vez implementado el resto de componentes del emulador se procedió a implementar el PSG. Como en los casos anteriores, se recurrió a una demo técnica para probar su correcto funcionamiento.

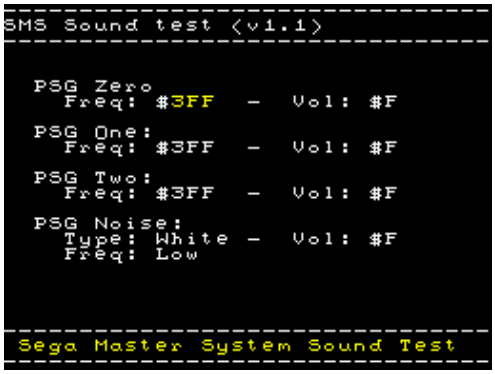


Fig. 52. Demo “SMS Sound Test” para verificar el PSG

Finalmente, con todo el emulador implementado y cada componente verificado por separado, se procedió a verificar todo en conjunto. Para ello se ejecutaron varios juegos comerciales en el emulador, comparándose la salida visual, música, sonidos y movimientos con los mismos juegos ejecutados en una Master System real, anotándose las diferencias o errores observados para posteriormente hacer una traza hasta localizar los fallos detectados.

Tras comprobar el correcto funcionamiento de una veintena de juegos se dio por finalizada la fase de verificación y pruebas, considerándose como válido el emulador desarrollado.

Capítulo 10. Conclusiones.

1. Objetivos cumplidos.

Al término de este trabajo final de carrera se han cumplido prácticamente todos los objetivos planteados en un principio, a excepción de la posibilidad de interactuar en red desde dos equipos conectados a una misma instancia del emulador.

Se ha llevado a cabo la implementación orientada a objetos, algo no muy común en este tipo de programas, resultando en un código más modular y reutilizable.

Se han incluido cuatro modos distintos de representar la imagen resultante, combinando filtros gráficos de suavizado con reescalados. Se podría haber incorporado algún filtro adicional, pero éstos se han considerado suficientes.

La frecuencia de muestreo de sonido del emulador ha sido implementada de forma totalmente configurable. Aunque desde el propio emulador se limite a tres frecuencias seleccionables (la configuración más usada en cualquier programa de sonido), la función que inicializa el sonido acepta cualquier valor compatible con el PC donde se ejecute.

Tal y como se propuso, se ha incluido la posibilidad de cargar y salvar el estado de ejecución de la máquina virtual mediante *savestates*.

Por otro lado se ha incluido una sencilla interfaz gráfica de usuario para facilitar la configuración y uso del emulador, permitiendo su ejecución seleccionando opciones en ésta, sin necesidad de recordar los parámetros de línea de comandos.

Finalmente, se ha realizado un programa en modo gráfico que emula un procesador Z80 con un acceso total a toda su información interna (registros, biestables, mapa de memoria, etc.), usado para propósitos de depuración de la clase Z80, pero perfectamente usable de modo independiente para emular un procesador de este tipo.

2. Perspectiva.

Si bien en la elaboración de este trabajo final de carrera se ha implementado un emulador completo, podría tomarse como base para futuros proyectos en los que ampliar sus características o reutilizar componentes de éste.

Las primeras Master System japonesas incorporaban un chip de sonido FM que fue eliminado en posteriores modelos para abaratar costes. Su implementación es bastante complicada (usa muestras de instrumentos digitalizados en lugar de generar ondas modulables en amplitud como el PSG) y podría suponer una ampliación interesante.

Aunque en un primer momento se tomó como objetivo a conseguir, se ha comentado que no se ha incluido soporte para jugar en red. Una posible ampliación sería incorporar este soporte comunicando dos instancias del emulador en máquinas remotas mediante *sockets* TCP/IP, de modo que una haga de servidor recibiendo las entradas de ambas instancias y devuelva los códigos de operación de instrucciones a ejecutar a la instancia cliente.

Muchos ordenadores personales y videoconsolas de los años 80 y 90 compartían hardware con la Master System. Las clases implementadas en este proyecto podrían reutilizarse en emuladores de máquinas con componentes similares como el Z80 en el Amstrad CPC, Sinclair ZX-Spectrum o los distintos MSX; el VDP con ligeras modificaciones en los MSX o el PSG en el BBC Micro o la SEGA Megadrive.

Si bien el emulador implementado tiene un alto grado de compatibilidad con los programas diseñados para la Master System, hay algunos de ellos que no funcionan correctamente o no llegan a cargar. Una posible ampliación sería estudiar minuciosamente el código máquina de éstos para encontrar el porqué de su malfuncionamiento y realizar las modificaciones oportunas en el emulador para darles soporte.

Los controles usados en el emulador están prefijados y no son configurables. Sería interesante añadir en la interfaz gráfica de usuario la posibilidad de redefinir las teclas asociadas a los mandos de control.

Bibliografía.

1. Libros consultados.

"Programación del Z80 con ensamblador". Olivier Lepape. Paraninfo, 1985.

"Sistema Microcomputador basado en Z-80". P. Cobos Arribas y otros. EUIT Telecomunicación, 1991.

"Programación del Microprocesador Z-80". Peter R. Rony. Marcobo, 1984.

"Microprocesador Z-80, Programación e Interfaces". Peter R. Rony. Marcobo, 1984.

"Microcomputers and Microprocessors". John Uffenbeck. Prentice-Hall, 1985.

"Programación del Z80". Rodnay Zaks. Anaya, 1985.

"KDE 2 / Qt Programming Bible". Arthur Griffith. IDG Books Worldwide, 2001.

"Programación orientada a objetos con C++". Fco. Javier Ceballos. Ra-Ma, 2003.

"C/C++ Curso de Programación". Fco. Javier Ceballos. Ra-Ma, 2002.

"Análisis y diseño orientado a objetos de sistemas usando UML". Simon Bennet y otros. McGraw-Hill, 2006.

"Programación orientada a objetos". Jordi Brínquez y otros. Editorial UOC, 2007.

"Programación orientada a objetos con C++". E. Balagurusamy. McGraw-Hill, 2007.

"Study of the techniques for emulation programming". Víctor Moya del Barrio. FIB UPC, 2001.

2. Vínculos de Internet.

"Qt4 tutorial for absolute beginners".
http://sector.ynet.sk/qt4-tutorial/

"Programación en C++ con Qt bajo Entorno GNU/Linux". Martín Sande.
http://www.cellfrancescsabat.org/CELL/seccions/GNU%20Linux/Programació/C++%20%20%20Programación%20en%20C++%20usando%20Qt.pdf

"Guía de estilo de programación en C++". J. Baltasar García Perez-Schofield.
http://trevinca.ei.uvigo.es/~jgarcia/TO/guiaestilocpp.pdf

"Software Reference Manual for the SEGA Mark III Console". SEGA.
http://garzul.tonsite.biz/SMS/Doc/official_mk3_manual.pdf

"SMS/GG hardware notes". Charles MacDonald.
http://www.smspower.org/dev/docs/smstech-20021112.txt

"SEGA Master System Technical Information". Richard Talbot-Watkins.
http://www.smspower.org/dev/docs/richard.txt

"The Undocumented Z80 Documented". Sean Young.
http://www.myquest.nl/z80undocumented/z80-documented-v0.91.pdf

"SEGA Master System VDP Documentation". Charles MacDonald.
http://www.smspower.org/dev/docs/msvdp-20021112.txt

"SN76489 Notes". Maxim.
http://www.smspower.org/dev/docs/sound/SN76489-20030421.txt

Agradecimientos.

Desde aquí quisiera dar las gracias a las distintas personas sin las cuales seguramente este proyecto no podría haberse llevado a cabo:

Mis padres, por apoyarme todos estos años, hacer de mí la persona que soy y, en menor medida, ser quienes trajeron una Master System y mi primer ordenador personal a casa.

José Gabriel Pérez Díez, por aceptar desde el primer momento dirigir este trabajo final de carrera y por hacerme entender cómo funciona un compilador y los intérpretes.

Casi todos mis profesores de la Escuela Universitaria de Informática y Facultad de Informática de la Universidad Politécnica de Madrid, por hacerme comprender las asignaturas, proporcionarme parte de sus conocimientos y, sobre todo, estar siempre dispuestos a ayudar cuando lo he necesitado.

Charles MacDonald, Nicola Salmoria, Omar Cornut, Marat Fayzullin y tantos otros pioneros de la emulación por iniciar el camino y compartir sus conocimientos con todos los interesados en la materia.

Linus Torvalds y Richard M. Stallman, por sus contribuciones al proyecto GNU/Linux, la creación de la licencia GPL y la fomentación del software libre, vía indiscutible de conocimientos.

Apéndice A. La biblioteca QT4.

Qt4 es la última versión de la biblioteca Qt desarrollada por Trolltech.

Ésta es una biblioteca multiplataforma enfocada a la realización de interfaces gráficas de usuario. Qt presenta una interfaz común debajo de la cual hace uso de las APIs específicas de cada sistema operativo soportado (en la última versión Microsoft Windows, GNU/Linux y MacOSX), consiguiendo de este modo una gran eficiencia y una apariencia prácticamente igual, se ejecute en el sistema en que se ejecute.

Qt incorpora ciertas utilidades para facilitar el desarrollo de interfaces gráficas, resultando su confección visual ciertamente trivial, evitando al programador el tener que crear cada objeto en forma de código y así poder centrarse en el funcionamiento de los botones, menús u otros objetos soportados por esta biblioteca.

A pesar de esta sencillez, Qt es una biblioteca muy potente como lo demuestra el hecho de que el entorno de escritorio KDE de GNU/Linux haga uso de ella para todas las interfaces de usuario.

La elección de esta biblioteca y no otra para la interfaz de usuario de este proyecto ha sido tomada por los puntos especificados anteriormente: facilidad de uso, potencia, soporte multiplataforma y estar licenciada bajo una licencia libre GPL (General Public License).

Apéndice B. Las bibliotecas SDL.

Las bibliotecas SDL (acrónimo de Simple DirectMedia Layer) comprenden una serie de bibliotecas multimedia desarrolladas originalmente por Sam Lantinga, un miembro de la compañía Loki Software.

Estas bibliotecas son multiplataforma y están portadas a múltiples sistemas operativos (GNU/Linux, Windows, MacOSX, BeOS...), permitiendo que los programas desarrollados con ellas puedan ser recompilados para otras máquinas o sistemas operativos prácticamente sin realizar cambio alguno.

Las bibliotecas que forman SDL actúan como una capa de abstracción entre las funciones propias del sistema operativo huésped y el programador. Así una misma llamada a una función gráfica de SDL hará uso de la biblioteca DirectX, Xlib o Quartz dependiendo de si se ejecuta bajo Windows, GNU/Linux o MacOSX, respectivamente.

SDL está modularizado en varios subsistemas, permitiendo cada subconjunto de estas librerías proporcionar acceso a rutinas de video, audio, entradas de teclado y joysticks, temporizadores...

Finalmente, SDL está licenciado bajo una licencia libre de tipo LGPL (Lesser General Public License) permitiendo su uso libre siempre que se respete ésta.

Al igual que en el caso de las biblioteca Qt, la elección de estas bibliotecas para la representación multimedia del emulador ha sido tomada debido principalmente a sus características multiplataforma y licencia libre de uso, además de poseer bastante experiencia previa en su uso en otros desarrollos software.

Apéndice C. El simulador Z80Sim.

El componente más importante de un emulador, análogamente al de una máquina real, es el intérprete de su procesador principal. Por este motivo es conveniente asegurarse, en la medida de lo posible, de que su comportamiento sea lo más fiel al procesador real antes de empezar a desarrollar el resto de componentes del emulador.

Para la prueba y depuración del procesador se ha desarrollado el simulador Z80Sim, el cual desde una interfaz gráfica de usuario permite simular paso a paso el comportamiento de las principales funciones del procesador Z80.

En la siguiente figura se muestra la interfaz de usuario:

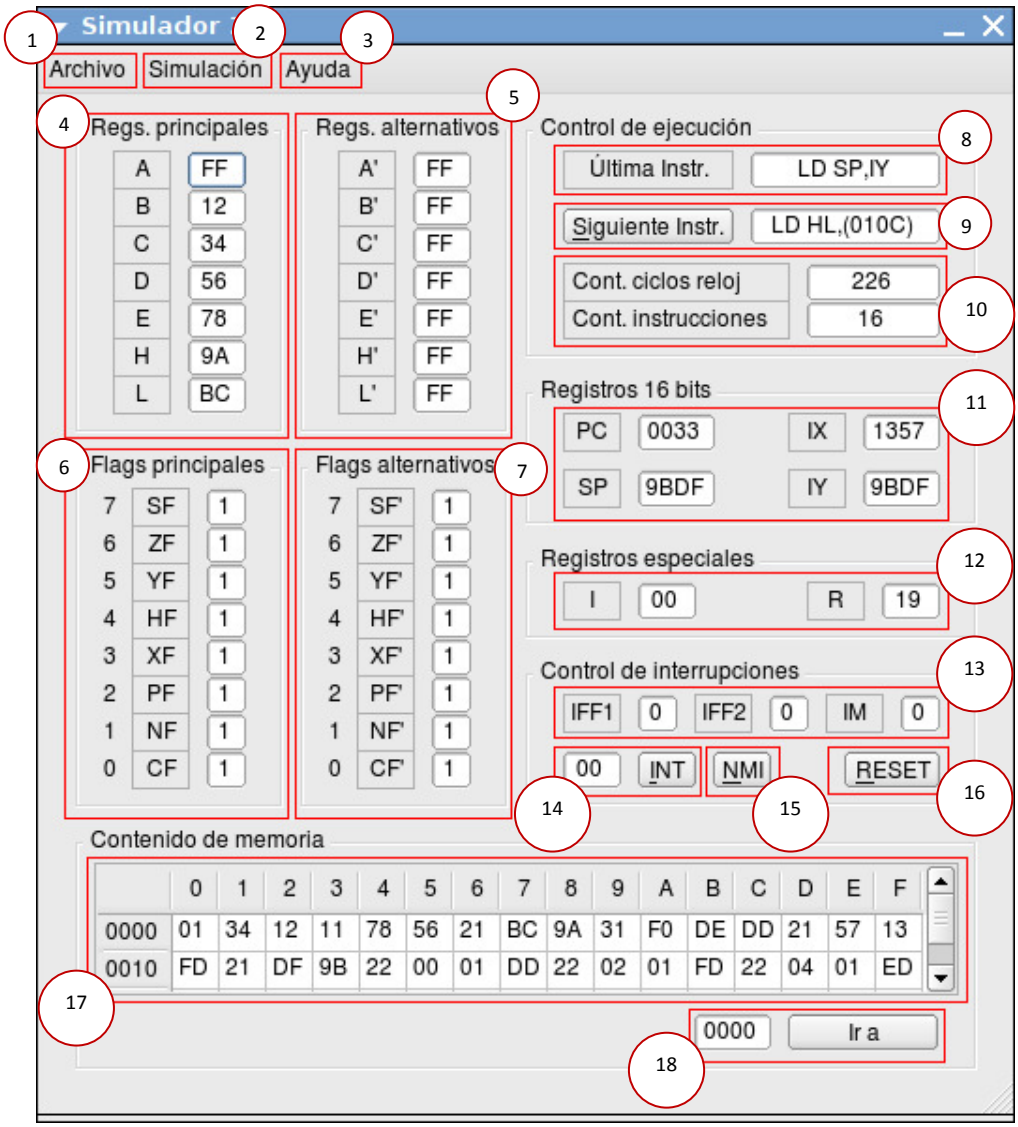


Fig. 53. Simulador Z80Sim

A continuación se detalla cada uno de los apartados de la misma:

1. Desde el menú Archivo se seleccionará el programa a cargar en el simulador para su ejecución. También contempla la opción de salir del simulador.
2. A través del menú Simulación se podrá ejecutar paso a paso el programa previamente cargado en el simulador.
3. En el menú Ayuda se tiene acceso a una pantalla de información sobre el autor del simulador.
4. En este apartado puede verse el contenido de cada uno de los registros individuales de propósito general del banco activo del Z80. Su valor viene expresado en hexadecimal.
5. En este apartado puede verse el contenido de cada uno de los registros individuales de propósito general del banco secundario del Z80. Su valor viene expresado en hexadecimal.
6. En este apartado aparecen desglosados de forma individual cada uno de los flags del registro F del banco activo del Z80.
7. En este apartado aparecen desglosados de forma individual cada uno de los flags del registro F del banco secundario del Z80.
8. En este apartado se muestra el mnemónico de la última instrucción ejecutada con sus parámetros correspondientes.
9. En este apartado se muestra el mnemónico de la próxima instrucción a ser ejecutada con sus parámetros correspondientes. Al pulsar en el botón denominado "Siguiente Inst." se ejecutará dicha instrucción.
10. En estos dos contadores se muestra el número de ciclos de reloj e instrucciones ejecutados hasta el momento.
11. Aquí se muestra el contenido de los registros de 16 bits del Z80 (contador de programa, puntero de pila, IX e IY). Su valor viene expresado en hexadecimal.
12. En este apartado se muestra el contenido de los registros I y R. Su valor viene expresado en hexadecimal.
13. En este apartado se muestra el contenido de los biestables IFF1 e IFF2 así como el modo de interrupción (IM) en el que se encuentra actualmente el Z80.
14. Para generar una interrupción de tipo enmascarable se introducirá en el cuadro de texto de la izquierda el valor del vector de interrupción (si fuera necesario para el modo de interrupción actual) y se pulsará el botón etiquetado como INT. Al ejecutar la siguiente instrucción se llevará a cabo el salto a la rutina adecuada.
15. Para generar una interrupción de tipo no enmascarable se pulsará el botón etiquetado como NMI. Al ejecutar la siguiente instrucción se llevará a cabo el salto a la rutina adecuada.
16. Este botón es usado para generar un *reset* hardware (no confundir con la instrucción RESET) del Z80.

17. En esta matriz se muestra el contenido de todas las direcciones de memoria direccionables por el Z80. Su valor viene expresado en hexadecimal y es posible editarlo haciendo *click* en la celda correspondiente y escribiendo el valor deseado.
18. Si bien es posible explorar toda la memoria usando la barra de desplazamiento que acompaña a las celdas, avanzar a posiciones lejanas puede ser lento o tedioso. Para evitar esto se ha implementado este apartado desde el que es posible saltar a cualquier dirección simplemente introduciendo la misma (en hexadecimal) en el campo de texto y pulsando el botón "Ir a".

Su uso es sumamente sencillo, tan solo hay que cargar el programa binario en código máquina del Z80 e ir pulsando el botón "siguiente" (o la tecla "s") para avanzar paso a paso.

Además de esta interfaz gráfica se ha implementado a través de la salida estándar una traza, instrucción a instrucción, donde se muestra en una línea de texto el valor de cada registro del procesador.

Apéndice D. Instrucciones detalladas del Z80.

1. Acrónimos y notación.

En el presente apéndice se detallarán de forma esquemática cada una de las instrucciones del repertorio del procesador Z80.

Para todas ellas se tendrán en cuenta los siguientes acrónimos:

- NB : Número de bytes de memoria ocupados por la instrucción.
- NC: Número de ciclos de máquina requeridos para la ejecución de la instrucción.
- NT: Número de T-estados de la máquina requeridos para la ejecución de la instrucción.
- Hex: Valor hexadecimal del código de operación de la instrucción.
- r, r' significa cualquiera de los registros A, B, C, D, E, H, L.
- p, p' significa cualquiera de los registros A, B, C, D, E, IX_H, IX_L.
- q, q' significa cualquiera de los registros A, B, C, D, E, IY_H, IY_L.
- dd_L, dd_H se refiere a los ocho bits inferiores o superiores del registro respectivamente.
- * significa instrucción oficialmente no documentada.

En cuanto a los flags se tendrá en cuenta la siguiente notación:

- • = este flag no es afectado.
- 0 = este flag se establece a cero.
- 1 = este flag se establece a uno.
- ↕ = este flag se establece de acuerdo al resultado de la operación.
- IFF_2 = se copia el valor del biestables 2 en este flag.

2. Grupo de instrucciones de carga de 8 bits.

Mnemónico	Operación simbólica	Flags S	Z	F5	H	F3	P/V	N	C	Cód. Oper. 76 543 210	Hex	N B	N C	N T	Comentarios
LD r, r'	r ← r'	•	•	•	•	•	•	•	•	01 r r'		1	1	4	r, r' Reg.
LD p, p'*	p ← p'	•	•	•	•	•	•	•	•	11 011 101 01 p p'	DD	2	2	8	000 B 001 C
LD q, q'*	q ← q'	•	•	•	•	•	•	•	•	11 111 101 01 q q'	FD	2	2	8	010 D 011 E
LD r, n	r ← n	•	•	•	•	•	•	•	•	00 r 110 ← n →		2	2	7	100 H 101 L
LD p, n*	p ← n	•	•	•	•	•	•	•	•	11 011 101 00 p 110 ← n →	DD	3	3	11	111 A p, p' Reg.
LD q, n*	q ← n	•	•	•	•	•	•	•	•	11 111 101 00 q 110 ← n →	FD	3	3	11	000 B 001 C 010 D
LD r, (HL)	r ← (HL)	•	•	•	•	•	•	•	•	01 r 110		1	2	7	011 E
LD r, (IX + d)	r ← (IX + d)	•	•	•	•	•	•	•	•	11 011 101 01 r 110 ← d →	DD	3	5	19	100 IX_H 101 IX_L 111 A
LD r, (IY + d)	r ← (IY + d)	•	•	•	•	•	•	•	•	11 111 101 01 r 110 ← d →	FD	3	5	19	q, q' Reg. 000 B
LD (HL), r	(HL) ← r	•	•	•	•	•	•	•	•	01 110 r		1	2	7	001 C
LD (IX + d), r	(IX + d) ← r	•	•	•	•	•	•	•	•	11 011 101 01 110 r ← d →	DD	3	5	19	010 D 011 E 100 IY_H
LD (IY + d), r	(IY + d) ← r	•	•	•	•	•	•	•	•	11 111 101 01 110 r ← d →	FD	3	5	19	101 IY_L 111 A
LD (HL), n	(HL) ← n	•	•	•	•	•	•	•	•	00 110 110 ← n →	36	2	3	10	
LD (IX + d), n	(IX + d) ← n	•	•	•	•	•	•	•	•	11 011 101 00 110 110 ← d → ← n →	DD 36	4	5	19	
LD (IY + d), n	(IY + d) ← n	•	•	•	•	•	•	•	•	11 111 101 00 110 110 ← d → ← n →	FD 36	4	5	19	
LD A, (BC)	A ← (BC)	•	•	•	•	•	•	•	•	00 001 010	0A	1	2	7	
LD A, (DE)	A ← (DE)	•	•	•	•	•	•	•	•	00 011 010	1A	1	2	7	
LD A, (nn)	A ← (nn)	•	•	•	•	•	•	•	•	00 111 010 ← n → ← n →	3A	3	4	13	
LD (BC), A	(BC) ← A	•	•	•	•	•	•	•	•	00 000 010	02	1	2	7	
LD (DE), A	(DE) ← A	•	•	•	•	•	•	•	•	00 010 010	12	1	2	7	
LD (nn), A	(nn) ← A	•	•	•	•	•	•	•	•	00 110 010 ← n → ← n →	32	3	4	13	
LD A, I	A ← I	↕	↕	↕	0	↕	IFF_2	0	•	11 101 101 01 010 111	ED 57	2	2	9	
LD A, R	A ← R	↕	↕	↕	0	↕	IFF_2	0	•	11 101 101 01 011 111	ED 5F	2	2	9	R es leído después de ser incrementado.
LD I, A	I ← A	•	•	•	•	•	•	•	•	11 101 101 01 000 111	ED 47	2	2	9	
LD R, A	R ← A	•	•	•	•	•	•	•	•	11 101 101 01 001 111	ED 4F	2	2	9	R es escrito después de ser incrementado.

Notas:
r, r' significa cualquiera de los registros A, B, C, D, E, H, L.
p, p' significa cualquiera de los registros A, B, C, D, E, IX_H, IX_L.
q, q' significa cualquiera de los registros A, B, C, D, E, IY_H, IY_L.
dd_L, dd_H se refiere a los ocho bits inferiores o superiores del registro respectivamente.
* significa instrucción oficialmente no documentada.

3. Grupo de instrucciones de carga de 16 bits.

Mnemónico	Operación simbólica	Flags S	Z	F5	H	F3	P/V	N	C	Cód. Oper. 76 543 210	Hex	N B	N C	N T	Comentarios
LD dd, nn	dd ← nn	•	•	•	•	•	•	•	•	00 dd0 001 ← n → ← n →		3	3	10	dd Par 00 BC 01 DE 02 HL 03 SP
LD IX, nn	IX ← nn	•	•	•	•	•	•	•	•	11 011 101 00 110 001 ← n → ← n →	DD 21	4	4	14	
LD IY, nn	IY ← nn	•	•	•	•	•	•	•	•	11 111 101 00 110 001 ← n → ← n →	FD 21	4	4	14	
LD HL, (nn)	L ← (nn) H ← (nn+1)	•	•	•	•	•	•	•	•	00 101 010 ← n → ← n →	2A	3	5	16	
LD dd, (nn)	dd_L ← (nn) dd_H ← (nn+1)	•	•	•	•	•	•	•	•	11 101 101 01 dd1 011 ← n → ← n →	ED	4	6	20	
LD IX, (nn)	IX_L ← (nn) IX_H ← (nn+1)	•	•	•	•	•	•	•	•	11 011 101 00 101 010 ← n → ← n →	DD 2A	4	6	20	
LD IY, (nn)	IY_L ← (nn) IY_H ← (nn+1)	•	•	•	•	•	•	•	•	11 111 101 00 101 010 ← n → ← n →	FD 2A	4	6	20	
LD (nn), HL	(nn) ← L (nn+1) ← H	•	•	•	•	•	•	•	•	00 100 010 ← n → ← n →	22	3	5	16	
LD (nn), dd	(nn) ← dd_L (nn+1) ← dd_H	•	•	•	•	•	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	DD	4	6	20	
LD (nn), IX	(nn) ← IX_L (nn+1) ← IX_H	•	•	•	•	•	•	•	•	11 011 101 00 100 010 ← n → ← n →	DD 22	4	6	20	
LD (nn), IY	(nn) ← IY_L (nn+1) ← IY_H	•	•	•	•	•	•	•	•	11 111 101 00 100 010 ← n → ← n →	FD 22	4	6	20	
LD SP, HL	SP ← HL	•	•	•	•	•	•	•	•	11 111 001	F9	1	1	6	
LD SP, IX	SP ← IX	•	•	•	•	•	•	•	•	11 011 101 11 111 001	DD F9	2	2	10	
LD SP, IY	SP ← IY	•	•	•	•	•	•	•	•	11 111 101 11 111 001	FD F9	2	2	10	
PUSH qq	SP ← SP - 1 (SP) ← qq_H SP ← SP - 1 (SP) ← qq_L	•	•	•	•	•	•	•	•	11 qq0 101		1	3	11	qq Par 00 BC 01 DE 10 HL 11 AF
PUSH IX	SP ← SP - 1 (SP) ← IX_H SP ← SP - 1 (SP) ← IX_L	•	•	•	•	•	•	•	•	11 011 101 11 100 101	DD E5	2	4	15	
PUSH IY	SP ← SP - 1 (SP) ← IY_H SP ← SP - 1 (SP) ← IY_L	•	•	•	•	•	•	•	•	11 111 101 11 100 101	FD E5	2	4	15	
POP qq	(SP) ← qq_L SP ← SP + 1 (SP) ← qq_H SP ← SP + 1	•	•	•	•	•	•	•	•	11 qq0 001		1	3	10	
POP IX	(SP) ← IX_L SP ← SP + 1 (SP) ← IX_H SP ← SP + 1	•	•	•	•	•	•	•	•	11 011 101 11 100 001	DD E1	2	4	14	

POP IY	(SP) ← IY_L SP ← SP + 1 (SP) ← IY_H SP ← SP + 1	•	•	•	•	•	•	•	•	11 111 101 11 100 001	FD E1	2	4	14	

4. Grupo de instrucciones Aritméticas y Lógicas de 8 bits.

Mnemónico	Operación Simbólica	S	Z	F5	H	F3	P/V	N	C	Cód. Oper. 76 543 210	Hex	NB	NC	NT	Comentarios
ADD A, r	A ← A + r	↕	↕	↕	↕	↕	V	0	↕	10 000 r		1	1	4	
ADD A, p*	A ← A + p	↕	↕	↕	↕	↕	V	0	↕	11 011 101 10 000 p	DD	2	2	8	
ADD A, q*	A ← A + q	↕	↕	↕	↕	↕	V	0	↕	11 111 101 10 000 q	FD	2	2	8	
ADD A, n	A ← A + n	↕	↕	↕	↕	↕	V	0	↕	11 000 110 ← n →		2	2	8	
ADD A, (HL)	A ← A + (HL)	↕	↕	↕	↕	↕	V	0	↕	10 000 110		1	2	7	
ADD A, (IX + d)	A ← A + (IX + d)	↕	↕	↕	↕	↕	V	0	↕	11 011 101 10 000 110 ← d →	DD	3	5	19	
ADD A, (IY + d)	A ← A + (IY + d)	↕	↕	↕	↕	↕	V	0	↕	11 111 101 10 000 110 ← d →	FD	3	5	19	
ADC A, s	A ← A + s + CY	↕	↕	↕	↕	↕	V	0	↕	001					
SUB A, s	A ← A - s	↕	↕	↕	↕	↕	V	1	↕	010					
SBC A, s	A ← A - s - CY	↕	↕	↕	↕	↕	V	1	↕	011					
AND s	A ← A AND s	↕	↕	↕	1	↕	P	0	0	100					
OR s	A ← A OR s	↕	↕	↕	0	↕	P	0	0	110					
XOR s	A ← A XOR s	↕	↕	↕	0	↕	P	0	0	101					
CP s	A - s	↕	↕	↕[1]	↕	↕[1]	V	1	↕	111					
INC r	r ← r + 1	↕	↕	↕	↕	↕	V	0	•	00 r 100		1	1	4	
INC p*	p ← p + 1	↕	↕	↕	↕	↕	V	0	•	11 011 101 00 p 100	DD	2	2	8	
INC q*	q ← q + 1	↕	↕	↕	↕	↕	V	0	•	11 111 101 00 q 100	FD	2	2	8	
INC (HL)	(HL) ← (HL) + 1	↕	↕	↕	↕	↕	V	0	•	00 110 100		1	3	11	
INC (IX + d)	(IX + d) ← (IX + d) + 1	↕	↕	↕	↕	↕	V	0	•	11 011 101 00 110 100 ← d →	DD	3	6	23	
INC (IY + d)	(IY + d) ← (IY + d) + 1	↕	↕	↕	↕	↕	V	0	•	11 111 101 00 110 100 ← d →	FD	3	6	23	
DEC m	m ← m - 1	↕	↕	↕	↕	↕	V	1	•	101					

r Reg. p Reg.
000 B 000 B
001 C 001 C
010 D 010 D
011 E 011 E
100 H 100 IX_H
101 L 101 IX_H
111 A 111 A

s es cualquiera de r, n, (HL), (IX+d), (IY+d), p, q como se ve para la instrucción ADD.

Los bits subrayados reemplazan los bits subrayados en el grupo de ADD.

q Reg.
000 B
001 C
010 D
011 E
100 IY_H
101 IY_L
111 A

m es cualquiera de r, p, q, (HL), (IX+d), (IY+d) como se muestra para la instrucción INC. DEC tiene el mismo formato y estados que INC, solo reemplaza 100 con 101 en el código de operación.

Notas:

[1] F5 y F3 son copiados del operando (s), no del resultado de (A-s).

El símbolo V en la columna del flag P/V indica que el flag P/V contiene el desbordamiento de la operación. De igual modo, el símbolo P indica paridad.

r significa cualquiera de los registros A, B, C, D, E, H, L.

p significa cualquiera de los registros A, B, C, D, E, IY_H, IY_L.

dd_L, dd_H se refieren respectivamente a los 8 bits inferiores o superiores del registro.

CY significa el biestable de acarreo.

* significa instrucción oficialmente no documentada.

5. Grupo de instrucciones Aritméticas de 16 bits.

Mnemónico	Operación Simbólica	S	Z	F5	H	F3	P/V	N	C	Cód. Oper. 76 543 210	Hex	N B	N C	N T	Comentarios
ADD HL, ss	HL ← HL + ss	•	•	↕[2]	↕[2]	↕[2]	•	0	↕[1]	00 ss1 001		1	3	11	ss Reg. 00 BC
ADC HL, ss	HL ← HL + ss + CY	↕[1]	↕[1]	↕[2]	↕[2]	↕[2]	V[1]	0	↕[1]	11 101 101 01 ss1 010	ED	2	4	15	01 DE 10 HL
SBC HL, ss	HL ← HL - ss - CY	↕[1]	↕[1]	↕[2]	↕[2]	↕[2]	V[1]	1	↕[1]	11 101 101 01 ss0 010	ED	2	4	15	11 SP
ADD IX, pp	IX ← IX + pp	•	•	↕[2]	↕[2]	↕[2]	•	0	↕[1]	11 011 101 00 pp1 001	DD	2	4	15	pp Reg.
ADD IY, rr	IY ← IY + rr	•	•	↕[2]	↕[2]	↕[2]	•	0	↕[1]	11 111 101 00 rr1 001	FD	2	4	15	00 BC 01 DE
INC ss	ss ← ss + 1	•	•	•	•	•	•	•	•	00 ss0 011		1	1	6	10 IX
INC IX	IX ← IX + 1	•	•	•	•	•	•	•	•	11 011 101 00 100 011	DD 23	2	2	10	11 SP
INC IY	IY ← IY + 1	•	•	•	•	•	•	•	•	11 111 101 00 100 011	FD 23	2	2	10	rr Reg. 00 BC
DEC ss	ss ← ss - 1	•	•	•	•	•	•	•	•	00 ss1 011		1	1	6	01 DE
DEC IX	IX ← IX - 1	•	•	•	•	•	•	•	•	11 011 101 00 101 011	DD 2B	2	2	10	10 IY 11 SP
DEC IY	IY ← IY - 1	•	•	•	•	•	•	•	•	11 111 101 00 101 011	FD 2B	2	2	10	

Notas:

El símbolo V en la columna del flag P/V indica que el flag P/V contiene el desbordamiento de la operación.
Las sumas de 16 bits son realizadas sumando primero los 8 bits inferiores de las dos palabras y después los 8 bits superiores.
[1] Indica que el flag es afectado por el resultado de 16 bits de la operación.
[2] Indica que el flag es afectado por la suma de 8 bits de la parte alta de las palabras.
CY significa el biestable de acarreo.

6. Grupo de instrucciones Aritméticas de Propósito General y Control de CPU.

Mnemónico	Operación Simbólica	S	Z	F5	H	F3	P/V	N	C	Cód. Oper. 76 543 210	Hex	N B	N C	N T	Comentarios
DAA	Convierte A en formato BCD.	↕	↕	↕	↕	↕	P	•	↕	00 100 111	27	1	1	4	
CPL	$A \leftarrow \overline{A}$	•	•	↕[1]	1	↕[1]	•	1	•	00 101 111	2F	1	1	4	Complemento a uno.
NEG[4]	A ← 0 - A	↕	↕	↕	↕	↕	V	1	↕	11 101 101 01 000 100	ED 44	2	2	8	Complemento a dos.
CCF	$CY \leftarrow \overline{CY}$	•	•	↕[1]	↕[2]	↕[1]	•	0	↕	00 111 111	3F	1	1	4	Complementa el flag de acarreo.
SCF	CY ← 1	•	•	↕[1]	0	↕[1]	•	0	1	00 110 111	37	1	1	4	
NOP	No operación	•	•	•	•	•	•	•	•	00 000 000	00	1	1	4	
HALT	Detiene la CPU	•	•	•	•	•	•	•	•	01 110 110	76	1	1	4	
DI[3]	$IFF_1 \leftarrow 0$ $IFF_2 \leftarrow 0$	•	•	•	•	•	•	•	•	11 110 011	F3	1	1	4	
EI[3]	$IFF_1 \leftarrow 1$ $IFF_2 \leftarrow 1$	•	•	•	•	•	•	•	•	11 111 011	FB	1	1	4	
IM 0[4]	Establece modo de interrupción 0	•	•	•	•	•	•	•	•	11 101 101 01 000 110	ED 46	2	2	8	
IM 1[4]	Establece modo de interrupción 1	•	•	•	•	•	•	•	•	11 101 101 01 010 110	ED 56	2	2	8	
IM 2[4]	Establece modo de interrupción 2	•	•	•	•	•	•	•	•	11 101 101 01 011 110	ED 5E	2	2	8	

Notas:

El símbolo V en la columna del flag P/V indica que el flag P/V contiene el desbordamiento de la operación. Análogamente el símbolo P indica paridad.
[1] F5 y F3 son una copia del bit 5 y 3 del registro A.er A
[2] H contiene el estado inicial del acarreo (después de la instrucción $H \leftrightarrow \overline{C}$)
[3] No se aceptan interrupciones directamente después de una instrucción DI o EI.
[4] Esta instrucción tiene otros códigos de operación no documentados.
CY significa el biestable de acarreo.

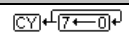
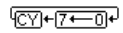
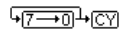
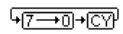
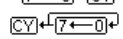
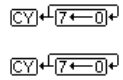
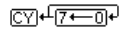
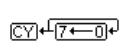
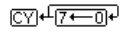
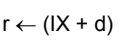
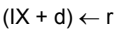
7. Grupo de instrucciones de Búsqueda, Intercambio y Transferencia.

Mnemónico	Operación Simbólica	Flags S	Z	F5	H	F3	P/V	N	C	Cód. Oper. 76 543 210	Hex	N B	N C	N T	Comentarios
EX DE, HL	DE ↔ HL	•	•	•	•	•	•	•	•	11 101 011	EB	1	1	4	
EX AF, AF'	AF ↔ AF'	•	•	•	•	•	•	•	•	00 001 000	08	1	1	4	
EXX	BC ↔ BC' DE ↔ DE' HL ↔ HL'	•	•	•	•	•	•	•	•	11 011 001	D9	1	1	4	
EX (SP), HL	(SP+1) ↔ H (SP) ↔ L	•	•	•	•	•	•	•	•	11 100 011	E3	1	5	19	
EX (SP), IX	(SP+1) ↔ IX_H (SP) ↔ IX_L	•	•	•	•	•	•	•	•	11 011 101 11 100 011	DD E3	2	6	23	
EX (SP), IY	(SP+1) ↔ IY_H (SP) ↔ IY_L	•	•	•	•	•	•	•	•	11 111 101 11 100 011	FD E3	2	6	23	
LDI	(DE) ← (HL) DE ← DE + 1 HL ← HL + 1 BC ← BC - 1	•	•	↕[1]	0	↕[2]	↕[3]	0	•	11 101 101 10 100 000	ED A0	2	4	16	
LDIR	(DE) ← (HL) DE ← DE + 1 HL ← HL + 1 BC ← BC - 1 repetir hasta: BC = 0	•	•	↕[1]	0	↕[2]	0	0	•	11 101 101 10 110 000	ED B0	2 2	5 4	21 16	si BC ≠ 0 si BC = 0
LDD	(DE) ← (HL) DE ← DE - 1 HL ← HL - 1 BC ← BC – 1	•	•	↕[1]	0	↕[2]	↕[3]	0	•	11 101 101 10 101 000	ED A8	2	4	16	
LDDR	(DE) ← (HL) DE ← DE - 1 HL ← HL - 1 BC ← BC - 1 repetir hasta: BC = 0	•	•	↕[1]	0	↕[2]	0	0	•	11 101 101 10 111 000	ED B8	2 2	5 4	21 16	si BC ≠ 0 si BC = 0
CPI	A - (HL) HL ← HL + 1 BC ← BC -1	↕[4]	↕[4]	↕[5]	↕[4]	↕[6]	↕[3]	1	•	11 101 101 10 100 001	ED A1	2	4	16	
CPIR	A - (HL) HL ← HL + 1 BC ← BC -1 repetir hasta: A = (HL) o BC = 0	↕[4]	↕[4]	↕[5]	↕[4]	↕[6]	↕[3]	1	•	11 101 101 10 110 001	ED B1	2 2	5 4	21 16	si BC ≠ 0 y A ≠ (HL). si BC = 0 o A = (HL)
CPD	A - (HL) HL ← HL - 1 BC ← BC -1	↕[4]	↕[4]	↕[5]	↕[4]	↕[6]	↕[3]	1	•	11 101 101 10 101 001	ED A9	2	4	16	
CPDR	A - (HL) HL ← HL - 1 BC ← BC -1 repetir hasta: A = (HL) o BC = 0	↕[4]	↕[4]	↕[5]	↕[4]	↕[6]	↕[3]	1	•	11 101 101 10 111 001	ED B9	2 2	5 4	21 16	si BC ≠ 0 y A ≠ (HL). si BC = 0 o A = (HL)

Notas:

[1] F5 es una copia del bit 1 de A + el ultimo byte transferido
[2] F3 es una copia del bit 3 de A + el ultimo byte transferido
[3] P/V flag es 0 si el resultado de BC - 1 = 0, en caso contrario P/V = 1.
[4] Estos flags son establecidos como en CP (HL)
[5] F5 es una copia del bit 1 de A - la última dirección comparada - H. H es como F después de la comparación.
[6] F3 es una copia del bit 3 de A - la última dirección comparada - H. H es como F después de la comparación.

8. Grupo de instrucciones de Rotación y Desplazamiento.

Mnemónico	Operación Simbólica	S	Z	F5	H	F3	P/V	N	C	Cód. Oper. 76 543 210	Hex	N B	N C	N T	Comentarios
RLCA		•	•	↕	0	↕	•	0	↕	00 000 111	07	1	1	4	
RLA		•	•	↕	0	↕	•	0	↕	00 010 111	17	1	1	4	
RRCA		•	•	↕	0	↕	•	0	↕	00 001 111	0F	1	1	4	
RRA		•	•	↕	0	↕	•	0	↕	00 011 111	1F	1	1	4	
RLC r		↕	↕	↕	0	↕	P	0	↕	11 001 011 00 000 r	CB	2	2	8	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
RLC (HL)		↕	↕	↕	0	↕	P	0	↕	11 001 011 00 000 110	CB	2	4	15	
RLC (IX + d)		↕	↕	↕	0	↕	P	0	↕	11 011 101 11 001 011 ← d → 00 000 110	DD CB	4	6	23	
RLC (IY + d)		↕	↕	↕	0	↕	P	0	↕	11 111 101 11 001 011 ← d → 00 000 110	FD CB	4	6	23	
LD r,RLC (IX + d)*	r ← (IX + d) RLC r (IX + d) ← r	↕	↕	↕	0	↕	P	0	↕	11 011 101 11 001 011 ← d → 00 000 r	DD CB	4	6	23	
LD r,RLC (IY + d)*	r ← (IY + d) RLC r (IY + d) ← r	↕	↕	↕	0	↕	P	0	↕	11 111 101 11 001 011 ← d → 00 000 r	FD CB	4	6	23	
RL m		↕	↕	↕	0	↕	P	0	↕	010					El formato de instrucción y los estados son los mismos que en RLC. Reemplaza 000 con el nuevo úmero.
RRC m		↕	↕	↕	0	↕	P	0	↕	001					
RR m		↕	↕	↕	0	↕	P	0	↕	011					
SLA m		↕	↕	↕	0	↕	P	0	↕	100					
SLL m*		↕	↕	↕	0	↕	P	0	↕	110					
SRA m		↕	↕	↕	0	↕	P	0	↕	101					
SRL m		↕	↕	↕	0	↕	P	0	↕	111					
RLD		↕	↕	↕	0	↕	P	0	•	11 101 101 01 101 111	ED 6F	2	5	18	
RRD		↕	↕	↕	0	↕	P	0	•	11 101 101 01 100 111	ED 67	2	5	18	

Notas:
El símbolo P en la columna del flag P/V indica que dicho flag contiene la paridad del resultado.
r significa cualquiera de los registros A, B, C, D, E, H, L.
* significa instrucción oficialmente no documentada.
CY significa el biestable de acarreo.

9. Grupo de instrucciones de Manipulación de bits.

Mnemónico	Operación Simbólica	S	Z	F5	H	F3	P/V	N	C	Cód. Oper. 76 543 210	Hex	N B	N C	N T	Comentarios
BIT b, r	$Z \leftarrow \overline{r_b}$	↕[1]	↕	↕[2]	1	↕[3]	↕[4]	0	•	11 001 011 01 b r	CB	2	2	8	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
BIT b, (HL)	$Z \leftarrow \overline{(HL)_b}$	↕[1]	↕	↕[2]	1	↕[3]	↕[4]	0	•	11 001 011 01 b 110	CB	2	3	12	
BIT b, (IX + d)[5]	$Z \leftarrow \overline{(IX + d)_b}$	↕[1]	↕	↕[2]	1	↕[3]	↕[4]	0	•	11 011 101 11 001 011 ← d → 01 b 110	DD CB	4	5	20	
BIT b, (IY + d)[5]	$Z \leftarrow \overline{(IY + d)_b}$	↕[1]	↕	↕[2]	1	↕[3]	↕[4]	0	•	11 111 101 11 001 011 ← d → 01 b 110	FD CB	4	5	20	
SET b, r	$r_b \leftarrow 1$	•	•	•	•	•	•	•	•	11 001 011 11 b r	CB	2	2	8	b Bit. 000 0 001 1 010 2 011 3 100 4 101 5 110 6 111 7
SET b, (HL)	$(HL)_b \leftarrow 1$	•	•	•	•	•	•	•	•	11 001 011 11 b 110	CB	2	4	15	
SET b, (IX + d)	$(IX + d)_b \leftarrow 1$	•	•	•	•	•	•	•	•	11 011 101 11 001 011 ← d → 11 b 110	DD CB	4	6	23	
SET b, (IY + d)	$(IY + d)_b \leftarrow 1$	•	•	•	•	•	•	•	•	11 111 101 11 001 011 ← d → 11 b 110	FD CB	4	6	23	
LD r,SET b, (IX + d)*	$r \leftarrow (IX + d)$ $r_b \leftarrow 1$ $(IX + d) \leftarrow r$	•	•	•	•	•	•	•	•	11 011 101 11 001 011 ← d → 11 b r	DD CB	4	6	23	
LD r,SET b, (IY + d)*	$r \leftarrow (IY + d)$ $r_b \leftarrow 1$ $(IY + d) \leftarrow r$	•	•	•	•	•	•	•	•	11 111 101 11 001 011 ← d → 11 b r	FD CB	4	6	23	
RES b, m	$m_b \leftarrow 0$ m ≡ r, (HL), (IX+d), (IY+d)	•	•	•	•	•	•	•	•	10					Para formar el nuevo código de operación reemplaza 11 de SET b, s con 10. Los flags y estados son los mismos.

Notas:

La notación m_b indica el bit b (0 a 7) de la palabra m.
Las instrucciones BIT son realizadas mediante un AND a nivel de bits.
[1] S es establecido si b = 7 y Z = 0.
[2] F5 es establecido si b = 5 y Z = 0.
[3] F3 es establecido si b = 3 y Z = 0.
[4] P/V es establecido como el flag Z.
[5] Esta instrucción tiene otros códigos de operación no documentados.
* significa instrucción oficialmente no documentada.

10. Grupo de instrucciones de Entrada / Salida.

Mnemónico	Operación Simbólica	Flags S	Z	F5	H	F3	P/V	N	C	Cód. Oper. 76 543 210	Hex	N B	N C	N T	Comentarios
IN A, (n)	A ← (n)	•	•	•	•	•	•	•	•	11 011 011 ← n →	DB	2	3	11	r Reg. 000 B
IN r, (C)	r ← (C)	↕	↕	↕	0	↕	P	0	•	11 101 101 01 r 000	ED	2	3	12	001 C 010 D
IN (C)* o IN F, (C)*	Solo afecta a los flags el valor es perdido.	↕	↕	↕	0	↕	P	0	•	11 101 101 01 110 000	ED 70	2	3	12	011 E 100 H 101 L
INI	(HL) ← (C) HL ← HL + 1 B ← B - 1	↕[1]	↕[1]	↕[1]	↕[3]	↕[1]	X	↕[2]	↕[3]	11 101 101 10 100 010	ED A2	2	4	16	111 A
INIR	(HL) ← (C) HL ← HL + 1 B ← B - 1 Repetir hasta que B = 0	0	1	0	↕[3]	0	X	↕[2]	↕[3]	11 101 101 10 110 010	ED B2	2 2	5 4	21 16	si B ≠ 0 si B = 0
IND	(HL) ← (C) HL ← HL - 1 B ← B - 1	↕[1]	↕[1]	↕[1]	↕[4]	↕[1]	X	↕[2]	↕[4]	11 101 101 10 101 010	ED AA	2	4	16	
INDR	(HL) ← (C) HL ← HL - 1 B ← B - 1 Repetir hasta que B = 0	0	1	0	↕[4]	0	X	↕[2]	↕[4]	11 101 101 10 111 010	ED BA	2 2	5 4	21 16	si B ≠ 0 si B = 0
OUT (n), A	(n) ← A	•	•	•	•	•	•	•	•	11 010 011 ← n →	D3	2	3	11	
OUT (C), r	(C) ← r	•	•	•	•	•	•	•	•	11 101 101 01 r 001	ED	2	3	12	
OUT (C), 0*	(C) ← 0	•	•	•	•	•	•	•	•	11 101 101 01 110 001	ED 71	2	3	12	
OUTI	(C) ← (HL) HL ← HL + 1 B ← B - 1	↕[1]	↕[1]	↕[1]	X	↕[1]	X	X	X	11 101 101 10 100 011	ED A3	2	4	16	
OTIR	(C) ← (HL) HL ← HL + 1 B ← B - 1 Repetir hasta que B = 0	0	1	0	X	0	X	X	X	11 101 101 10 110 011	ED B3	2 2	5 4	21 16	si B ≠ 0 si B = 0
OUTD	(C) ← (HL) HL ← HL - 1 B ← B – 1	↕[1]	↕[1]	↕[1]	X	↕[1]	X	X	X	11 101 101 10 101 011	ED AB	2	4	16	
OTDR	(C) ← (HL) HL ← HL - 1 B ← B - 1 Repetir hasta que B = 0	0	1	0	X	0	X	X	X	11 101 101 10 111 011	ED BB	2 2	5 4	21 16	si B ≠ 0 si B = 0

Notas:

El símbolo V en la columna del flag P/V indica que el flag P/V contiene el desbordamiento de la operación. Análogamente el símbolo P indica que contiene la paridad.

r significa cualquiera de los registros A, B, C, D, E, H, L.

[1] el flag es afectado por el resultado de B ← B - 1 como en DEC B.

[2] N es una copia del bit 7 del último valor de la entrada (C).

[3] este flag contiene el acarreo de (((C + 1) AND 255) + (C))

[4] este flag contiene el acarreo de (((C - 1) AND 255) + (C))

* significa instrucción oficialmente no documentada.

11. Grupo de instrucciones de Salto.

Mnemónico	Operación Simbólica	Flags S	Z	F5	H	F3	P/V	N	C	Cód. Oper. 76 543 210	Hex	N B	N C	N T	Comentarios
JP nn	PC ← nn	•	•	•	•	•	•	•	•	11 000 011 ← n → ← n →	C3	3	3	10	ccc Condición 000 NZ 001 Z
JP cc, nn	si cc es cierto, PC ← nn	•	•	•	•	•	•	•	•	11 ccc 010 ← n → ← n →		3	3	10	010 NC 011 C 100 PO
JR e	PC ← PC + e	•	•	•	•	•	•	•	•	00 011 000 ← e - 2 →	18	2	3	12	101 PE 110 P 111 M
JR ss, e	si ss es cierto PC ← PC + e	•	•	•	•	•	•	•	•	00 ss 000 ← e - 2 →		2 2	3 2	12 7	si ss es cierto si ss es falso
JP HL	PC ← HL	•	•	•	•	•	•	•	•	11 101 001	E9	1	1	4	
JP IX	PC ← IX	•	•	•	•	•	•	•	•	11 011 101 11 101 001	DD E9	2	2	8	ss Condición 111 C
JP IY	PC ← IY	•	•	•	•	•	•	•	•	11 111 101 11 101 001	FD E9	2	2	8	110 NC 101 Z
DJNZ e	B ← B - 1 si B ≠ 0 PC ← PC + e	•	•	•	•	•	•	•	•	00 010 000 ← e - 2 →	10	2 2	2 3	8 13	100 NZ si B = 0 si B ≠ 0

Notas: e es un número con signo en complemento a dos dentro del rango <-126, 129>
e - 2 en el código de operación proporciona un número efectivo de PC + e como si el PC se incrementara en 2 antes de sumarle el número e.

12. Grupo de instrucciones de Llamada y Retorno.

Mnemónico	Operación Simbólica	Flags S	Z	F5	H	F3	P/V	N	C	Cód. Oper. 76 543 210	Hex	N B	N C	N T	Comentarios
CALL nn	SP ← SP - 1 (SP) ← PC_H SP ← SP - 1 (SP) ← PC_L PC ← nn	•	•	•	•	•	•	•	•	11 001 101 ← n → ← n →	CD	3	5	17	
CALL cc, nn	si cc es cierto SP ← SP - 1 (SP) ← PC_H SP ← SP - 1 (SP) ← PC_L PC ← nn	•	•	•	•	•	•	•	•	11 ccc 100 ← n → ← n →		3 3	3 5	10 17	si cc es falso si cc es cierto
RET	PC_L ← (SP) SP ← SP + 1 PC_H ← (SP) SP ← SP + 1	•	•	•	•	•	•	•	•	11 001 001	C9	1	3	10	
RET cc	si cc es cierto PC_L ← (SP) SP ← SP + 1 PC_H ← (SP) SP ← SP + 1	•	•	•	•	•	•	•	•	11 ccc 000		1 1	1 3	5 11	si cc es falso si cc es cierto cc Condic. 000 NZ 001 Z 010 NC 011 C 100 PO 101 PE 110 P 111 M
RETI[2]	PC_L ← (SP) SP ← SP + 1 PC_H ← (SP) SP ← SP + 1	•	•	•	•	•	•	•	•	11 101 101 01 001 101	ED 4D	2	4	14	
RETN[1,2]	PC_L ← (SP) SP ← SP + 1 PC_H ← (SP) SP ← SP + 1 IFF_1 ← IFF_2	•	•	•	•	•	•	•	•	11 101 101 01 000 101	ED 45	2	4	14	
RST p	SP ← SP - 1 (SP) ← PC_H SP ← SP - 1 (SP) ← PC_L PC ← p	•	•	•	•	•	•	•	•	11 t 111		1	3	11	t p 000 0h 001 8h 010 10h 011 18h 100 20h 101 28h 110 30h 111 38h

Notas: [1] Esta instrucción tiene otros códigos de operación no documentados.
[2] La instrucción también efectúa IFF_1 ← IFF_2

Código fuente.

1. Introducción.

El código fuente ha sido organizado en varios directorios, uno por cada clase básica usada. En la figura 54 se muestra en forma de árbol esta estructura de archivos y directorios.

Cada clase está dispuesta en su correspondiente directorio, pero dos de ellas, las correspondientes al PIO y Memoria, se usarán como clases genéricas desde las que heredarán las clases específicas usadas en el emulador gSMS (PIOSMS y MemSMS) y en el simulador Z80Sim (PIOSim y MemSim) debido a sus características concretas.

El contenido de los directorios se corresponde con:

- gSMS: emulador de Master System desarrollado.
- gui: interfaz gráfica de usuario y selector de programas.
- joysticks: clase para manejar los dispositivos controladores.
- mem: clase genérica del sistema de memoria.
- misc: tipos definidos que se usarán en el resto de archivos.
- pio: clase genérica del sistema de entrada/salida.
- psg: clase del generador de sonido programable.
- vdp: clase del procesador de video.
- z80: clase del procesador Zilog Z80.
- z80Sim: simulador z80Sim.

Finalmente cabe remarcar que los archivos *moc_xxxxx.cpp* y *ui_xxxxx.h* son generados automáticamente por el editor visual Qt Designer a partir de la representación gráfica realizada en éste.

```
TFC
|
|---gSMS
|       gSMS.cpp
|       Mastersystem.h
|       Mastersystem.cpp
|       MemSMS.h
|       MemSMS.cpp
|       PIOSMS.h
|       PIOSMS.cpp
|
|---gui
|       gSMSGui.h
|       gSMSGui.cpp
|       main.cpp
|       moc_gSMSGui.cpp
|       smsrom_good.h
|       smsrom_good.cpp
|       ui_gui.h
|
|---joysticks
|       Joysticks.h
|       Joysticks.cpp
|
|---mem
|       Memoria.h
|       Memoria.cpp
|
|---misc
|       tipos.h
|
|---pio
|       PIO.h
|       PIO.cpp
|
|---psg
|       PSG.h
|       PSG.cpp
|
|---vdp
|       VDP.h
|       VDP.cpp
|
|---z80
|       Z80.h
|       Z80.cpp
|
|___z80Sim
        main.cpp
        MemSim.h
        MemSim.cpp
        moc_Z80Sim.cpp
        PIOSim.h
        PIOSim.cpp
        ui_z80sim.h
        Z80Sim.h
        Z80Sim.cpp
```

Fig. 54. Estructura de archivos del código fuente.